Development of a Graphical Regression Test Tool for use in 3D Virtual Environments

Title	Virtual Environment Test Procedure
Written by	Bjørn Grønbæk & Brian Horn
Deliverydate	February 9, 2009
Course	Master Thesis

Abstract

Modern 3D graphical virtual environments are used in many areas, especially in games and simulators. Testing a graphical virtual environment is usually done by manual process, where a programmer or tester performs a repeating pattern of use and observes the application responses. With the increased uses of games and simulators, the need for a cheaper, faster, and more accurate testing procedure has become apparent. This thesis evaluates existing test procedures, and presents a step by step procedure for testing virtual environments. Test tools for supporting the use of the procedure is designed and implemented to confirm the validity of the procedure, and tests with the procedure and tools are done on a selection of sample applications developed using Delta3D. The process is evaluated, based on the test results, and it is found that it is practical method for testing virtual environments.

Preface

This report is the result of a thesis study conducted by Bjørn Grønbæk and Brian Horn towards obtaining the Master's Degree in Software Systems Engineering. The material presented in this thesis is a followup on the preliminary study described in [Horn and Grønbæk, 2008]. This thesis has been written under supervision of Ulrik Pagh Schultz, Associate Professor, Ph.D. The work was carried out in the period September 2008 to February 2009.

The Maersk Mc-Kinney Moller Institute for Production Technology
University of Southern Denmark, Odense
9th of February 2009

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 IFAD . . . 3
- 1.2 Problem Analysis . . . 4
- 1.3 Reading Guide . . . 4

2 Testing Virtual Environments **5**
- 2.1 GUI Testing . . . 5
  - 2.1.1 Motivation . . . 5
  - 2.1.2 Test Oracles . . . 6
  - 2.1.3 Difficulties in GUI Testing . . . 6
  - 2.1.4 GUI Test Cases . . . 8
- 2.2 Regression Testing . . . 9
  - 2.2.1 Definition . . . 9
  - 2.2.2 Motivation . . . 9
  - 2.2.3 Regression Testing in Practice . . . 10
- 2.3 Testing User Interfaces in Virtual Environments . . . 12
  - 2.3.1 User Interfaces in Virtual Environments . . . 12
  - 2.3.2 Background and Motivation . . . 13
  - 2.3.3 Automated Test Runner Design . . . 14
  - 2.3.4 Automated Test Runner Implementation . . . 16

3 Automated Graphical Regression Testing for Virtual Environments **18**
- 3.1 Goal and Intentions . . . 18
- 3.2 GUI Testing Virtual Environments . . . 19
  - 3.2.1 Oracle Information in Virtual Environments . . . 19
  - 3.2.2 Oracle Procedures in Virtual Environments . . . 21
  - 3.2.3 Test Oracles in Virtual Environments . . . 22
- 3.3 Regression Testing Virtual Environments . . . 23
  - 3.3.1 Testing Procedure Evaluation for VE's . . . 23
- 3.4 Procedure and Support Tools for Testing Virtual Environments . . . 24
  - 3.4.1 Testing Procedure . . . 25
  - 3.4.2 Support Tools . . . 29
- 3.5 Architecture . . . 31
  - 3.5.1 Delta3D Virtual Environment Application . . . 31
  - 3.5.2 Image Collector . . . 32
  - 3.5.3 GM VETS Tool . . . 32

3.5.4 Image Analyzer . . . 32
3.5.5 Test Reporter . . . 32
3.5.6 Support Tools . . . 32
3.6 Case Study . . . 33

4 A Testing Framework for Delta3D 34
4.1 Framework Design and Implementation . . . 34
4.1.1 Framework Design . . . 35
4.1.2 Organization of Frameworks . . . 36
4.1.3 Conceptual Model . . . 37
4.1.4 Framework Instantiation . . . 37
4.1.5 Framework Guidelines . . . 38
4.2 Delta3D Framework Architecture . . . 38
4.2.1 Libraries In Delta3D . . . 38
4.2.2 AI In Delta3D . . . 41
4.2.3 Translation and Rotation in Delta3D . . . 42
4.2.4 Delta3D Framework Evaluation . . . 43
4.3 Delta3D Game Manager Architecture . . . 44
4.3.1 Background and Motivation . . . 44
4.3.2 GM Framework Organization . . . 45
4.3.3 GM Framework Functionality . . . 46
4.3.4 Framework Instantiation with the GM . . . 50
4.4 GM Virtual Environment Test Support Tool . . . 51
4.4.1 Design . . . 52
4.4.2 Testing with a Delta3D Application . . . 54
4.4.3 Conclusion . . . 57
4.5 Delta3D AAR Architecture . . . 58
4.5.1 AAR Components . . . 58
4.5.2 Using AAR as a Test Support Tool . . . 61
4.5.3 Conclusion . . . 65

5 Testing 3D Graphics and Simulation Engines 66
5.1 Game Engines . . . 66
5.1.1 Camera Model . . . 67
5.1.2 Culling and Clipping . . . 67
5.1.3 Rasterization . . . 68
5.1.4 Animation . . . 69
5.1.5 Level Of Detail . . . 70
5.1.6 Terrain . . . 72
5.2 Comparing Models Using Multiple Views . . . 73
5.2.1 Rendering Parameters . . . 74
5.3 Scene Graphs . . . 76
5.3.1 Scene Composition And Management . . . 77
5.3.2 Representation Of Scene Graphs . . . 77
5.3.3 Open Scene Graph . . . 79
5.3.4 Scene Graph Matching . . . 79
5.4 Image Collector . . . 79
5.4.1 Multiple Views and Platonic Solids . . . 80
5.4.2 Interface to Delta3D . . . 80
5.4.3 Structure of the Image Collector . . . 82

5.4.4 Using the Image Collector in a Delta3D Application . . . 85
5.4.5 Testing the Image Collector . . . 89

6 Image Regression Testing 92
6.1 Basic Concepts . . . 92
6.2 Image Metrics . . . 93
6.2.1 Image Definitions And Notations . . . 94
6.2.2 Measures Based On Pixel Differences . . . 94
6.2.3 Correlation Based Measures . . . 95
6.2.4 Using Metrics in the Image Analyzer . . . 97
6.3 Visual Perception Of Images . . . 97
6.3.1 Contrast . . . 98
6.3.2 Acuity . . . 99
6.3.3 Object Border . . . 99
6.3.4 Color . . . 100
6.3.5 Perceptual Motivated Metrics . . . 101
6.4 The Image Metric Analyzer Component . . . 102
6.4.1 Design . . . 102
6.4.2 Implementation . . . 103
6.4.3 Test . . . 103
6.4.4 Conclusion . . . 107
6.5 Test Reporter Tool . . . 108
6.5.1 Test Case Selection and Execution . . . 108
6.5.2 Test Reporter . . . 112
6.5.3 Conclusion . . . 113

7 Perspectives 115
7.1 Related Work . . . 115
7.1.1 ANN's as Automated Oracles . . . 116
7.2 Future Work . . . 117
7.3 Conclusion . . . 118
7.3.1 Tool Development . . . 118
7.3.2 Problem Analysis . . . 119
7.3.3 In General . . . 120

A Installation and Configuration Guide 121
A.1 Delta3D And Visual Studio . . . 121
A.1.1 Compiling And Building Delta3D . . . 122
A.1.2 Delta3D Development Environment In Visual Studio . . . 123
A.2 OpenSceneGraph And Visual Studio . . . 125
A.3 Configuring Cruise Control .NET for C++ . . . 127
A.3.1 Cruise Control .NET . . . 127
A.3.2 NAnt . . . 130

B Spatial Transformations in Delta3D 133
B.1 Position . . . 133
B.2 Rotation . . . 134
B.3 Euler Angles from Rotation Matrix . . . 136
B.4 Finding the corresponding angles of h . . . 136
B.5 Finding the corresponding angles of r . . . 137

B.6 The two solutions 137
B.7 Special case 137

C The Platonic Solids **139**
C.1 Tetrahedron 139
C.2 Hexahedron 140
C.3 Octahedron 141
C.4 Icosahedron 142
C.5 Dodecahedron 144

List of Figures

2.1 Traditional GUI in a desktop application 12
2.2 GUI in a virtual environment 13
2.3 Test manager design 15
2.4 System classes 16
2.5 Test execution 17

3.1 "Traditional" GUI embedded in a virtual environment 19
3.2 Optional caption for list of figures 20
3.3 Test Case library structure 27
3.4 Component diagram showing the architecture of the regression test tool. 31

4.1 Whitebox and blackbox relationship 36
4.2 Library hierarchy in Delta3D. 39
4.3 Game Manager Framework - Selected classes 45
4.4 Game Manager set diagram 47
4.5 Game Manager message passing 48
4.6 Message Class Design 49
4.7 Game Actor class diagram 50
4.8 VETS class diagram 52
4.9 The HoverTank sample application 54
4.10 The GMTank sample application class diagram 55
4.11 Optional caption for list of figures 56
4.12 Optional caption for list of figures 57
4.13 Optional caption for list of figures 58
4.14 Optional caption for list of figures 59
4.15 AAR Logger Components 60
4.16 ServerLoggerComponent Class Diagram 61
4.17 LogController Class Diagram 62
4.18 AAR allows pausing and stopping replays 62
4.19 AAR allows skipping to keyframes 63
4.20 AAR allows changing the internal time scale 63
4.21 GameManager message path 64

5.1 View frustum showing a polygon projection from world space to screen plane. 68
5.2 Levels of detail and the view volume. 70

5.3 An front-view image of a 3D model of a rabbit. The position and orientation of the camera only captures the front of the model. . 74
5.4 Example of a 3D model of a rabbit seen from 12 camera different camera positions. The viewpoints and view orientations are distributed uniformly. 74
5.5 The Small Rhombicuboctahedron. The 24 vertices of this uniform polyhedron are used as the viewpoints surrounding an object in the off-line quality evaluation in the work carried out by Peter Lindstrom. 75
5.6 Scene graph programming models hides underlying graphics APIs and graphics rendering and display devices from the developer. . 77
5.7 The structure of a scene graph represented as a tree. 78
5.8 Class diagram showing the structure of the Image Collector. Only the most important attributes and methods are shown. 82
5.9 Animated soldier in town. (a) Normal view. (b) Navigation edges shown. 86
5.10 Town model. (a) seen from a top camera position. (b) Navigation edges shown. 87
5.11 Optional caption for list of figures 91

6.1 Geometry in perspective projection. 93
6.2 Conditional contrast - illusion of brightness. 98
6.3 Shading and brightness affect how a scene is perceived by the human eye. (a) Shadow across a board makes the color of the squares look different. (b) The gray bars show that the two squares have the same color. 99
6.4 The Ebbinghaus illusion. 99
6.5 RGB color space. 100
6.6 Image Analyzer test 1 104
6.7 Image Analyzer test 2 105
6.8 Image Analyzer test 3 106
6.9 Image Analyzer test 4 107
6.10 Image Analyzer test 5 108
6.11 The difference in position and rotation is calculated by the image analyzer and presented in a ”difference image”. 109
6.12 Report Generator and the WalkingSoldier test 112
6.13 Report Generator and the GMTank test 113
6.14 Report Generator and the GMTank test 2 114

7.1 ANN evaluation phase 117

A.1 Setting up ”Additional Include Directories” in Visual Studio. . . 124
A.2 Setting up ”Runtime Library” in Visual Studio. 125
A.3 Setting up ”Precompiled Headers” in Visual Studio. 126
A.4 Setting up ”Additional Library Directories” in Visual Studio. . . 127
A.5 Setting up ”Additional Dependencies” in Visual Studio. 128
A.6 Including link libraries in Visual Studio. 129
A.7 The Cruise Control .NET web dashboard. 130
A.8 The Cruise Control .NET CCTray main window. 131

B.1 A position vector in Delta3D. 134
B.2 Default coordinate system in Delta3D. 135

C.1 The tetrahedron is one of the five Platonic solids with four vertices, six edges, and four faces. 139
C.2 The hexahedron is one of the five Platonic solids with eight vertices, twelve edges, and six faces. 141
C.3 The octahedron is one of the five Platonic solids with six vertices, twelve edges, and eight faces. 141
C.4 The icosahedron is one of the five Platonic solids with twelve vertices, thirty edges, and twenty faces. 142
C.5 The dodecahedron is one of the five Platonic solids with twenty vertices, thirty edges, and twelve faces. 144

List of Tables

3.1 Steps in software GUI testing 22
3.2 Steps in software regression testing 23
3.3 Procedure test steps 25

6.1 Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes, which are completely identical. 105
6.2 Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes, where the soldier carries a weapon and where he does not. 106
6.3 Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes where the soldier carries a helmet and where he does not. 107
6.4 Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes where the soldier is present and where he does not. 108
6.5 Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes where the soldier is oriented differntly. 109

A.1 CruiseControl.NET installation prerequisites 128

Chapter 1

Introduction

This chapter presents an introduction to the thesis project. First, a motivation for developing a regression test tool for use in 3D virtual environments, is given. Secondly, the vision of the thesis work is presented. Finally, a reading guide is given.

1.1 Motivation

Graphical simulators, Virtual Reality (VR) applications, and computer games are being developed for many different fields and domains. Although developed for different reasons, a common element in most of these application, is that they use virtual environments as user interfaces. Such user interfaces are created as 3D graphical virtual environments, where navigation and interaction with the underlying application logic is handled differently compared to traditional 2D graphical user interfaces. This naturally affects the development and implementation of these applications. Development of VR applications, simulators, and computer games are complex processes, requiring developers to both handle the low-level mechanics of general software development, such as architecture, patterns dependencies etc, as well as high-level human-computer interaction interfaces. As usual, with any development project of a certain size, the chance of getting the product right on the first try is considered an impossible idea, and an iterative process of development and evaluation is needed.

Evaluation and validation in a graphical virtual environment development project can be done in a number of areas, for example system performance, usability and correctness. These areas can be evaluated in the usual way, for example usability can be evaluated formally through qualitative user testing at certain intervals, and system performance can be evaluated by the developers during development. Correctness can be evaluated by testing the application for expected responses, using different software testing techniques and a structured testing approach. The internal components can for example be tested with automated unit tests, and the correctness of a human-computer interface in a graphical application can be evaluated by doing manual tests of the user interface.

Manual testing of user interface correctness consists of following a basic and predefined pattern using the application, and then evaluating a number of

responses by the program. A typical pattern might be:

1. Start the application, which includes setting the needed configuration options of the program.

2. Interact with the application until the application is in the state that should be tested.

3. Perform the interaction that is to be tested, or evaluated, and observe the application's responses.

Such an pattern can become very time demanding, since it involves a real person interacting with the application in real time. The whole process is inherently error prone due to the way an exact pattern must be followed manually every time for the results to make sense. As the application evolves and becomes more complex, the amount of features to test increases, and the process of performing such a manual test process only worsen and becomes more and more in-practical to perform. Additionally, the increased use of virtual environment applications in professional fields, such as military training simulators, has only increases the demands of the testing process, by requiring thorough testing of all features, and accompanying documentation of tests and results.

Some automated testing techniques are already available in graphical application development, such as unit testing, which can be applied to the code in a normal way, and automated using one of the many existing unit testing tools. Some special considerations for using standard testing tools in virtual environment application development can be portability requirements, a large amount of third party libraries, in the form of for example game engines, and in general lack of what could be called a testing culture in development of virtual environment applications.

Similar to the situation with unit testing, some testing techniques and tools are available for testing human-computer interaction interfaces, which can be used as long as the interfaces are in a standardized form, such as GUIs. Interfaces in the form of GUIs are normally distributed as an collection of graphical widgets, such as buttons, sliders etc. in a library. GUIs can be tested by analyzing the interface's widget connections into the underlying code, and invoking methods in that code, as if a user had interacted with the widget. Even though such a process is possible, testing GUIs are often associated with difficulties. One is that the above description of testing is based on the underlying assumption that the connection between the GUI and the code is already correct when tests are executed, which might not be the case. Other difficulties arise from the fact that many GUIs have an infinite input path, which allows the user to press almost any buttons in almost any order, which results in a huge number of tests to guaranty correctness under all circumstances. Finally, the way many GUIs are implemented today, using graphical tools available through IDEs, makes testing of GUIs by capture/replay tools and increasingly difficult process. Since the GUI may be refactored and redesigned easily with such a tool, GUIs often change their layout throughout the implementation process, rendering recorded mouse clicks at specific coordinates invalid.

Simulators and VR environments do often, in addition to traditional GUI elements, make use of human-computer interfaces which are much more complex than the standard GUIs of most applications. In a graphical environment

users interact with the application not only through standardized widgets, but also through objects in the graphical environment itself. This might be by touching, rotating or moving objects is the user's perceived environment, but also by simply moving the user's view point around. Such interaction is not readily implemented in the form of widget libraries, but instead objects in the environment has access to, and uses, the application code directly. Naturally, this makes the interfaces very hard to test in an automated way, since objects in the environment and the users actions are not limited to a predefined set of possibilities as in GUIs. Testing is therefore bound to be undertaken by real users, or testers, in a manual way.

As described above, the user interface in a graphical environment, simulator or VR, are not limited to the standardized widgets found in a library. Since interaction with an user interface usually is a two-way process, the need for testing, not only user input, but also application output arises. In a typical GUI application that uses widgets, such output might be text ares being filled out, dialog-boxes being opened etc, but the situation is not as clear in a fully graphical environment. In a simulation output, or responses, from the application would typically also be graphical objects. For example moving a certain object (a lever or a button) might result in the appearance of other graphical objects (a door or a piece of equipment) as response. The application logic of such an operation can be tested in the usual way by creating unit test for all involved code, but the actual verification of objects in the environment are left to be done in a manual way by a tester.

1.1.1 IFAD

Before starting this thesis a preliminary investigation titled "Investigation of Elements for an Automated Test Tool for Virtual Environments" [Horn and Grønbæk, 2008], was carried out. The purpose of the investigation was to analyze the principles used in automated testing of virtual environments, in order to determine the research direction of this thesis. During this initial work, we cooperated with IFAD TS A/S, which is a Danish company specialized in developing training simulations. We were introduced to the IFACTS (IFAD Forward Air Controller Training Solution) simulator, a military training simulator designed to tactical training of fighter pilots and military ground personnel. The environment and surroundings in this simulator are modelled as a 3D virtual battlefield, where users interact with the simulator through standard I/O interfaces like keyboard and mouse - similar to 3D computer games. During our collaboration with IFAD, several problems in regression testing of graphical virtual environments were discussed. IFAD particularly wished to automate the process of validating the correctness of graphical scene objects in the IFACTS simulator. Their main problem, was that if changes were made to virtual environment objects and these changes were defective, they could propagate undetected to the final release. Additionally, issues in introducing a structured test process in an existing development project were discussed.

The collaboration with IFAD stopped after the preliminary study, and the thesis, described in this report, presents a more general and theoretical investigation of regression testing in virtual environment applications. The work carried out in the preliminary study is described in [Horn and Grønbæk, 2008], which also contains specific details about IFAD and the IFACTS simulator.

1.2 Problem Analysis

In this project we which to develop an automated test tool for the execution of graphical regression tests in virtual environments. The project will focus on procedures for performing graphical regression tests. The focus includes:

1. Implementation of image processing and image analysis methods, and the evaluation of these methods value as a tool for comparing scenes from 3D environments.

2. Evaluation of the feasibility of scene graph comparison, in the context of graphical regression testing.

3. The design and implementation of a set of tools for automation of graphical regression testing, and the combination of those tools with existing test automation tools.

The procedure developed will be done in a general fashion, and as such should be suitable for use in any graphical virtual environment. The design and evaluation of test support tools will be based on a general software framework analysis, but will be implemented with the Delta3D framework as platform.

1.3 Reading Guide

The following presents a short presentation of the purpose of each chapter in this report.

- Chapter 1: Gives an introduction to the report, including motivation and vision.

- Chapter 2: Introduces the domain of the thesis including definitions and terminologies used in regression testing of virtual environments.

- Chapter 3: Presents an analysis of the special requirements necessary for testing user interfaces in virtual environments, including a description of the overall features contained in the test tool which has been developed in the thesis.

- Chapter 4: Presents the central Delta3D framework extensions which have been designed and implemented in the thesis.

- Chapter 5: Describes the work done to create an image collector module to extend the Delta3D framework.

- Chapter 6: Presents details on using image processing in regression testing of virtual environments, including the design and implementation of an image analyzer module.

- Chapter 7: Summarizes the study performed and presents future research areas.

Chapter 2

Testing Virtual Environments

This chapter introduces several topics that are considered fundamental for understanding the background and need for the development of a test tool for virtual environments. First, a section on testing graphical user interfaces including the difficulties involved when creating GUI tests. The second section explains the concept of regression testing, some issues with regression testing in modern software engineering, and finally a suggestion for a process that can be used when creating regression tests. The third section gives a short introduction to why testing of GUI's in desktop software differs from testing the user interfaces of applications that use virtual environments. Finally, a review of a proposed system for testing user interfaces in virtual reality applications is presented.

2.1 GUI Testing

The purpose of this section is to introduce the techniques used when testing graphical user interfaces, in particular the concept of test oracles. A motivation for using GUI testing is given, and several issues that make GUI testing particular challenging in modern software development are presented. Finally, a number of common steps in a GUI testing process is listed and described.

2.1.1 Motivation

Most applications and clients in client server systems present their functionality to users through a GUI [Gerrard, 1997]. According to [Memon and Soffa, 2003] as much as half of all code lines in (2003) software systems are GUI code, making GUIs pervasive in most software. As with any other part of a software system, it is important to verify the functionality of the GUI. Maybe even especially so, since all user interaction with the program is bound to pass through the GUI layer of the system. The normal method to ensure correct behavior of the GUI code is not different from that of any other part of the system: comprehensive and thorough testing.

2.1.2 Test Oracles

Testing GUIs are done by creating a set of test cases and execute the test cases on the GUI being tested. The result of the execution is inspected using test oracles, which determines if the GUI behaves as expected.

[Xie and Memon, 2007] gives short introduction to the concept of test oracles. A test oracle contains two parts: First, oracle information which is the expected output of a test performed on a GUI. Secondly, an oracle procedure which takes care of comparing the actual output of the test with the oracle information. Oracles can be of different types, that is, they can use different things for oracle information and procedure. For example an oracle testing a spread sheet application could be in one configuration where it checks all fields in the application, or it could just check one field. The information contained in the oracle would depend on the type of test being performed. Similarly, the oracle information depends on how the oracle procedure is intended to behave, since it could function in several ways. It could be assigned to check that the output is exactly identical to the information, or it could be assigned to check that the output falls within a certain range of what is given in the oracle information. These two information and two procedure types combined, adds up to four types of oracles.

There are four approaches to creating test oracles that is currently being used. The first and most commonly used is the manual oracles, where a human tester interacts with the GUI, clicking buttons, writing in text fields and opening dialogs etc., and visually inspects the result. A second approach is the use of capture/replay tools. Creating an oracle with capture/replay is done in two phases: First a tester interacts with the program and visually asserts that the program is behaving as expected. The objects in the GUI used to assert the correctness is captured, along with the interactions the tester performed. Later the interactions can be replayed and the "assert-areas" checked to see if the GUI is working correctly. The third approach is to use asserts programmed in a test tool, like JUnit, and letting a test developer write explicit test cases that cover the GUI functionality. Finally, some test oracles are created by using hard coded test harnesses, simulating the input from a user interacting with the GUI, and call underlying application logic as if the program had been invoked from the GUI.

2.1.3 Difficulties in GUI Testing

Even though GUI testing is just as an important aspect of testing a complete software system as any other test discipline, GUI testing remains an ad hoc process. This is a consequence of the fact that GUI testing is a considerably more difficult discipline than traditional testing, and is a very labor intensive process. According to [Memon, 2002] traditional software testing may account for as much as 60% of software development costs, and GUI testing is considered to be even more resource demanding. [Gerrard, 1997] lists several reasons why GUI testing is more complicated than traditional testing:

- **Event driven:** For each GUI many possible user inputs can occur. Any menu, button field etc. of a GUI may receive the next user input. The user is not limited to a particular sequence of inputs, and might very likely not even be limited to the currently active window. At any time the

user could activate an event outside of the current window, by invoking another window in the application, or perhaps somewhere outside of the application, for example an OS window. This basically means that at almost any event might follow another particular event, and no prior order can be computed or expected. [Gerrard, 1997] calls this particular problem the *infinite path* problem.

- **Unsolicited events:** Some events might not be user generated, but instead be the result of external input to the system. Examples include pop-up windows notifying of an offline printer, messages from the OS and "focus stealing" from other applications, like instant messenger clients. Basically, this means that even if the user is somehow restricted to a non-infinite path of events, this might be ruined by new event paths being introduced by outside factors.
- **Object orientation:** Even though object-oriented programming is traditionally seen as simplifying development and testing, this might not be quite the result in the GUI testing field. The elements of a GUI maps very well to objects, so almost every element of a GUI is likely to be an object. Those objects are likely to each have their own methods and a (large) set of properties and values, such as text, text size, font, color, width etc. This means that even simple GUIs may have a very large number of methods and property sets, and many of those properties are likely to be dynamically changed as a result of user input.
- **Window management:** Most applications implement the standard window management functionality expected by users: resizing, moving, minimizing and closing. These events are generally handled by the OS, but the effects these events might have on the application must still be handled by the application itself. This for example, includes transaction management if a window is unexpectedly closed, when fields in the GUI have been filled out.

Additional Pitfalls

[Memon, 2002] describes several, what he calls pitfalls, in GUI testing, when compared to traditional testing.

Test Coverage Due in part, to the infinite path problem, the traditional way of using code coverage as a measure of the quality of test coverage, is not really viable. Even a small amount of code might have a large number of events, and a better criteria is probably to ensure that all combinations of events are covered. But the number of events and in particular the number of combinations make it impossible to test all such combinations. This means that at some point it is necessary for a test designer to decide on a particular subset of all possible tests, which then undermines the code coverage criteria, and instead puts it into the hands of the test designer to select an appropriate amount of tests, and appropriate parts of the application to be tested.

Test Oracles In GUI testing, and in traditional testing, a test oracle is used to verify the results of a test execution. But in contrast to traditional testing,

it might not be enough to execute a complete test case and then invoke the test oracle to compare the results with the expected results. In GUI testing it is often necessary to invoke the oracle after each step in the test case, since a step might render the GUI in an incorrect state, for example changing the active window. If this happens, it is very likely that every following step in the test case will also result in wrong results, meaning that the rest of the test execution is pointless. In traditional software testing such a wrong state could have be detected by the use of exceptions, but that is not an option in GUI testing. If the GUI testing is continued, in spite of a wrong state being entered, it is also possible that at some point a correct state is reached again. This is likely to prove problematic, since the final step in the test case may be correct, even though one or more steps during the execution actually failed.

Regression Testing Finally, a popular test method is regression testing, which involves running and re-running a number of test cases to discover the introductions of errors in a program. GUI testing presents special problems when used in regression testing. Regression testing often involves changes in the design of certain elements of the code. Changing the design of a GUI might very likely make the test cases designed for that particular GUI fail, even though no actual error may have been introduced. This is a consequence of the very nature of GUI testing, where the test input and output are often connected to the layout of the window being tested. This means that a very small change to a window, that might not affect the functionality of the application at all, can still render a large amount of the generated test cases obsolete. These test cases will fail, not because the application being tested is suddenly error prone, but because the precise input/output relation of the application has changed.

2.1.4 GUI Test Cases

Even though GUI testing is different from traditional testing, many testing techniques from other testing methodologies still applies to GUI testing. Both for traditional testing and GUI testing, the typical test process could include the following steps:

1. Before designing the tests, a test coverage criteria should be found. This criteria includes what to measure and what the measurement goal should be set to. The test criteria is used to ensure that relevant areas of an application is tested.

2. Test input must be generated for the test cases. The input should reflect the software specifications in the documentation. In GUI testing the input can be every way the user can interact with the GUI, typically based on uses cases. For example a test input could be every step needed to perform an open, edit and save use case in a document editor.

3. Expected output for a test case must be generated and stored. The expected output should be able to identify whether a GUI is behaving as expected, or if something has failed. Some outputs could be window positions, text in title bars or active buttons.

4. Execute the test cases and compare output with expected output using an oracle.

5. Check if the test output matches the test coverage criteria.

In addition to the previous steps a number of principles also known from traditional testing should be applied to GUI testing:

1. Categorize different errors into various types, and design tests specifically for each type of error. This allows test reuse when testing for errors in a specific category.

2. Separate complex errors into simpler types, so it is possible to reuse tests written for these types of errors.

3. Use a layered test strategy, with simplest tests at the bottom layer. This serves as basis of integration tests, by building "trusted layers" which other components can be integrated in.

2.2 Regression Testing

The purpose of this section is to give an overview of regression testing as a tool in software engineering. This section describes the concept of regression testing, explaining why and when it is appropriate to perform regression testing, and what bugs regression testing are supposed to identify. A motivation for performing regression testing, rooted in economical and some historical facts, is presented. Finally, a number of common steps in a regression testing cycle is listed and briefly described.

2.2.1 Definition

Regression testing is a software testing technique intended to ensure that code modification does introduce new errors into already tested code [Memon, 2002]. Regression testing is an important part of a modern software development life cycle. Regression testing is a testing process used to ensure that a modified program still meets its requirements, and that new errors have not been introduced during changes. An error, or bug, introduced during a change in the software is called a regression bug, and the change is not necessarily confined to code changes, but could also include hardware upgrades or changes to time settings etc. A common method for implementing regression testing is to rerun tests that have previously be run, and which are known to have run successfully, and see if a change has re-introduced an already fixed or handled bug. Regression testing is always recommended, since previous fixes are often lost through bad revision control, or are reintroduced through so-called fragile fixes, which only fix a general problem in a specific situation.

2.2.2 Motivation

According to [Wahl, 1999] it is estimated that 70% of the total cost of developing software are expended during the maintenance phase of a programs lifetime, and the longer a program exists, the more the cost of maintenance is increased. Regression testing is done to reduce the total cost of the software development, including maintenance. In addition [Wahl, 1999] points out

that regression testing is not only about economic concerns, but that regression testing also helps make software work correctly and reliably, which benefits both customers and developers. [Onoma et al., 1998] writes that in general regression testing is useful in all software development and maintenance, but for some types of companies it may be even more important. Companies with the following characteristics are reported to have especially good use of regression testing:

- Companies developing similar products, or a family of products, since test cases can easily be reused.
- Companies developing mission or safety critical software which need to be tested and re-tested frequently.
- Companies maintaining large program that exist over extended periods of time. Regression testing can serve as a sanity check for such software.
- Companies developing software that are under constant and changing development, for example as a result of fast market evolution.
- Companies that do not employ formal, or even semi-formal, development processes.
- Companies that utilize no other software inspection as part of the quality assurance techniques.

Regression testing is often automated and according to [Onoma et al., 1998] some software companies develop complete in-house regression testing tools, to automate the regression testing process. Since the most common method of implementing regression testing is to re-run existing tests, the number of test cases will increase over time, and a manual approach to performing and re-performing all tests are in many cases impractical. Automated regression testing also allows for easy scheduling of testing, such as at every commit/compile, every night or perhaps weekly if the test suite is large. Some companies try to minimize the number of test cases that are rerun in each regression test, by only running test cases that aim specifically at the changed code, but this is not a critical issue for all companies, and is therefore seen as optional by [Onoma et al., 1998]. Only running a subset of the existing test cases are called selective regression testing [Wahl, 1999], and only applies when there is no changes in the software specifications involved in the maintenance. Many tools, such as Make or NAnt, are capable of helping with selective regression testing, by only re-compiling parts of the code that has been changed, and only run the tests associated with that code.

2.2.3 Regression Testing in Practice

Even though software development houses use a variety of software development methodologies like waterfall, unified process, XP etc., [Onoma et al., 1998] reports that most use a very similar process in regression testing, following a number of steps as shown in the following list:

Modification request: The process starts out with a *modification request*. According to [Wahl, 1999] the modification request is usually received in

the maintenance phase, and can be either adaptive or perfective or it can be corrective. Corrective regression testing can be applied without changing the available test cases, since there are no changes in the specifications of the software. If the specifications are changed, progressive regression testing must be applied, which requires that new test cases are designed.

Software artifact modification: The software artifact, which apart from the actual source code may include documentation and configuration files, are changed to meet the modification request.

Test case selection: After modification, the relevant test cases to be run, must be selected. This could be all available test cases, but could also be a selected number of test cases. It is possible that *test case validation* is performed during this activity.

Test execution: After test case selection the tests are executed, usually in an automated fashion.

Failure identification: *Failure identification* is done by inspecting the results of the test case execution with the expected output, which is often part of the automated execution. If there is an incongruence between the results expected, and the results obtained, the test fails, else the test is considered passed. Inspecting the failed test case involves first re-validating the test, since it may not reflect the actual test case due to changed specifications. All failed test cases should be re-validated before the code is inspected, and only if the test case is valid, should the code be inspected. It is possible that both code and test were faulty, so the test case must be re-executed.

Fault identification: If the source code was found, or just suspected, to be faulty the code must be inspected during the *fault identification* activity. The exact component, version etc. under which the fault occurs, along with the exact modification causing the fault is found.

Fault mitigation: The final activity is to actually correct the fault.

[MSDN, 2008] recommends that developers should build a library of all tests written, and run these tests at every build resulting in a new version. They suggest that even tough selective regression testing is possible, it is better to err on the side of caution, and run to many tests rather than to few. All tests that are automated should always be run, as well as all manual tests that involve boundary conditions and timing. They also recommend that a regression testing library should not just contain tests that are know to have discovered bugs, but instead just as many tests as possible. If the test library becomes too large to run properly it should be inspected, and redundant and unnecessary tests removed. A typical problem is duplicate test cases, where lots of tests have been written to identify a bug in the code. After the bug has been eliminated, only the general test case that could help discover the bug should be included.

2.3 Testing User Interfaces in Virtual Environments

The purpose of this section is to give some background information in the field of testing graphical user interfaces in virtual environments. The section provides some background information on why an UI of a virtual environment must be treated differently than the traditional UI of an desktop application. An explanation as to what this means for testing, and why UI testing in virtual environments is more difficult is presented, while at the same time some motivation for doing UI testing is given. A suggestion for a design and implementation given in other literature is reviewed.

2.3.1 User Interfaces in Virtual Environments

In this project traditional graphical user interfaces (GUIs) and 3D graphical interfaces embedded in virtual environments are distinguished. A traditional GUI, is taken to mean an user interface as normally expected from a standard desktop program. An example of a GUI is shown in figure 2.1, which is a standard window created using Microsoft Windows library components. A GUI

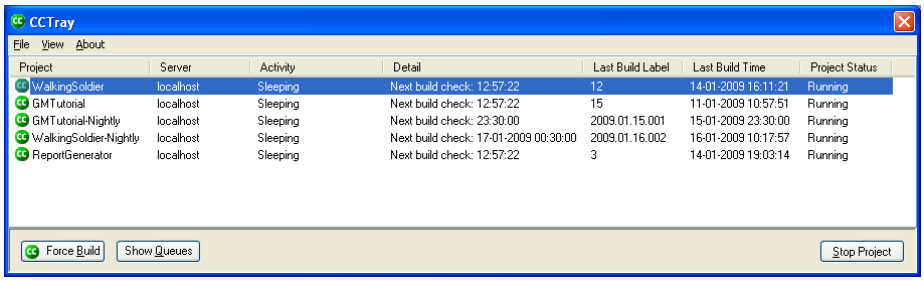


Figure 2.1: Traditional GUI in a desktop application

has standard components, such as buttons and text fields, and are normally distributed as a library.

3D graphical interfaces, or just graphical interface, are not standard components, but instead custom developed objects, in a virtual environment, that the user is able to interact with. The interaction with the application happens in much the same way as in a GUI, by binding the graphical object to the underlying code and invoking methods. The distinguishing factor, is that graphical interface objects are not part of a standard library, and the binding between user interface and application logic is custom developed, with the risk of introducing errors. Figure 2.2 shows an example of a graphical interface from a Delta3D application. Even though there is a difference between the two types of user interfaces described, the use of a graphical interface in a virtual environment does not exclude the use of GUI elements in the same environment. Typically, a simulation could present both a graphical environment, where the user is able to interact with one or more graphical objects, as discussed above, but also present traditional GUI elements in the same environment. For example in the form of a text area for writing chat messages, and buttons for sending messages. In figure 2.2 an example of such use is the text fields shown in the GUI. These labels are

Figure 2.2: GUI in a virtual environment

part of a more traditional GUI library that is available in Delta3D, but are also at the same time part of the graphical interface of the virtual environment.

2.3.2 Background and Motivation

[Bierbaum et al., 2003] writes that virtual environments are being developed and used in a wide range of fields and domains today. Developing a virtual environment (VE) system often requires handling both the low-level details associated with general development, as well as the complexity of developing an immersive interface for the user of the application. Many of the software engineering techniques used today use an iterative development process, alternating between development and evaluation. This is even more true when developing VE applications, where the complexity of the interface developed makes it infeasible to get a product correct in the first development cycle.

[Bierbaum et al., 2003] divides the evaluation of a VE application into four areas:

- System performance - evaluated by developers when new features are added.
- Usability - evaluated both by the developers on a daily basis and formally by user tests.
- Value - evaluated both by the developers on a daily basis and formally by user tests.

- Correctness - evaluated with systematic tests.

The last evaluation criteria is of special interest to developers of VE applications, since the correctness is more difficult to test in a VE application, than in a desktop application. Testing correctness in a VE application is primarily done by automated tests of internal components, typically using unit tests, and by performing manual tests of the user interface in the application. A manual test requires the tester to run the application and perform the actual interaction with the application, as the user would do in an use case scenario. For each use case being tested, the tester is required to perform several steps:

1. Perform configuration - This can for example include network and other application specific configuration. Configuration could also include preparing a test case, by creating appropriate folders, moving files and in general preparing test artifacts.

2. Provide interaction - The test case usually has to be performed when the application is in a specific state. The tester manually brings the application to this state, by interacting with the application through the interface. For example moving the player character around, and interacting with different simulation actors.

3. Perform and evaluate test - When the application is in the sought state, the tester must perform and evaluate the test case. This could for example include interacting with a certain object in the game, like a door, and then observing that the game arrives at a certain state, with the door open for example, as a result of the test case being executed.

This is naturally a very time consuming task, and there is a risk that errors are introduced when the tester manually prepares the application in this way. When the complexity of the application increases as result of more and more iterations being completed, the work being spent testing in this fashion increases.

[Bierbaum et al., 2003] writes that automated testing of VE application do take place in development today, but is usually limited to classic testing techniques, like unit testing. Unit testing can indeed be used to confirm correctness of internal data, like algorithms and classes, but are harder to use when it comes to testing user interaction. To create an automated testing system for VE applications, it is necessary to create an automated user interaction setup, which would focus not on functional tests, like unit testing does, but rather on the graphical interface, and on the interaction between users and the application.

2.3.3 Automated Test Runner Design

In [Bierbaum et al., 2003], a technique for applying automated testing in virtual reality applications is suggested. The authors propose a design for a testing framework, that uses a test runner that maintains a list of tests that it manages. Each of these tests externally monitor the state of the VR application, while the application is being controlled by pre-recorded input data. When the application reaches a state that requires testing, the test becomes active and checks the application for the correct state. An proposed design of such a system is shown in figure 2.3. Here the design shows the central `TestRunner` which keeps tracks of a number of test cases. Each `TestCase` monitors the state of the VR application,

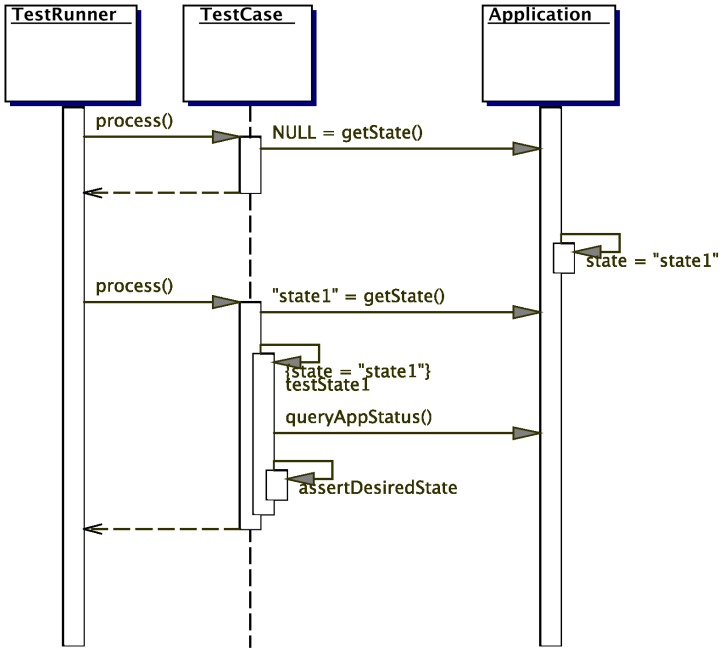


Figure 2.3: Test manager design

while the application is running. Figure 2.3 does not show how the application is controlled, but it is assumed that the application is replaying pre-recorded input. When the `TestCase` discovers the application to be in a state that is requiring testing, it can verify that the internal state of the application is as expected.

In order to automate the testing procedure, input is provided to the application in a way that responds exactly to how a user would interact with it. This is done by recording all device inputs during an application usage scenario. At selected stages during the recording, the input is stamped and user-defined aspects of the state of the application are stored. This log of the input data and application state is used later as the basis of a test case. When the VR application has to be tested, a test case that monitors the application is created. When the test case begins its execution, it loads the recorded input into the system. Periodically, the test case queries the input playback system and the application to get the current state of both. When the application reaches some checkpoint, the test case verifies that the application state matches the previously stored state. If the application state is incorrect, then the test case signals the test runner, and the test runner alerts the user the failure. When all tests have completed execution, the application consults the test runner for the test results. At this point, the user reviews the results and makes changes to fix test failures.

2.3.4 Automated Test Runner Implementation

[Bierbaum et al., 2003] implements a simple testing module that can handle and run automated tests for a sample application. The implementation is specifically written for the VR Juggler suite, but the concepts should be applicable in other VE's, including Delta3D. Figure 2.4 shows the core classes used in the test

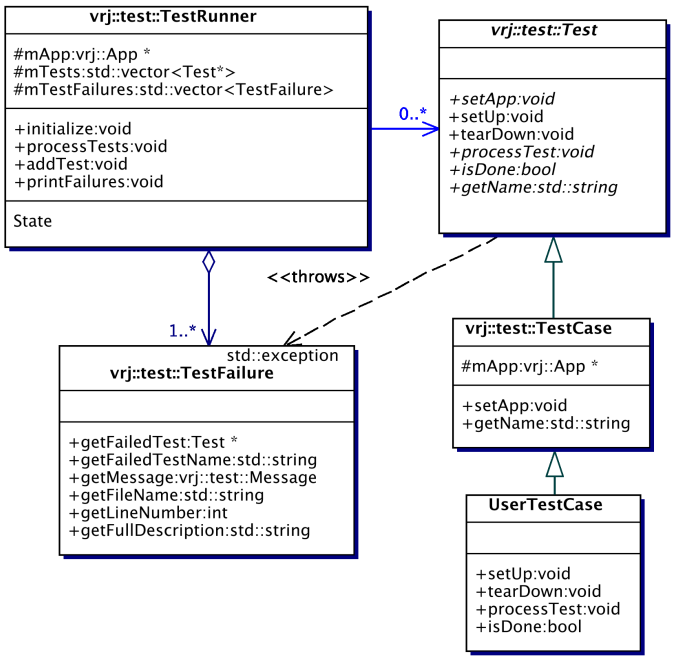


Figure 2.4: System classes

tool. The `TestRunner` class binds the test modules together, and is responsible for communication with the VE application. The `Test` class holds a number of test cases that must be checked during the execution of a test, and creates objects from the TestFailure class if necessary, which it would then pass on to the `TestRunner`. Figure 2.5 shows a sequence diagram depicting the execution of a test run with the design described by [Bierbaum et al., 2003]. The VE invokes first the test runner and then a test case, which is added to the test runner. The test runner then takes over the control with the VE application, and executes the necessary tests and checks for the test case.

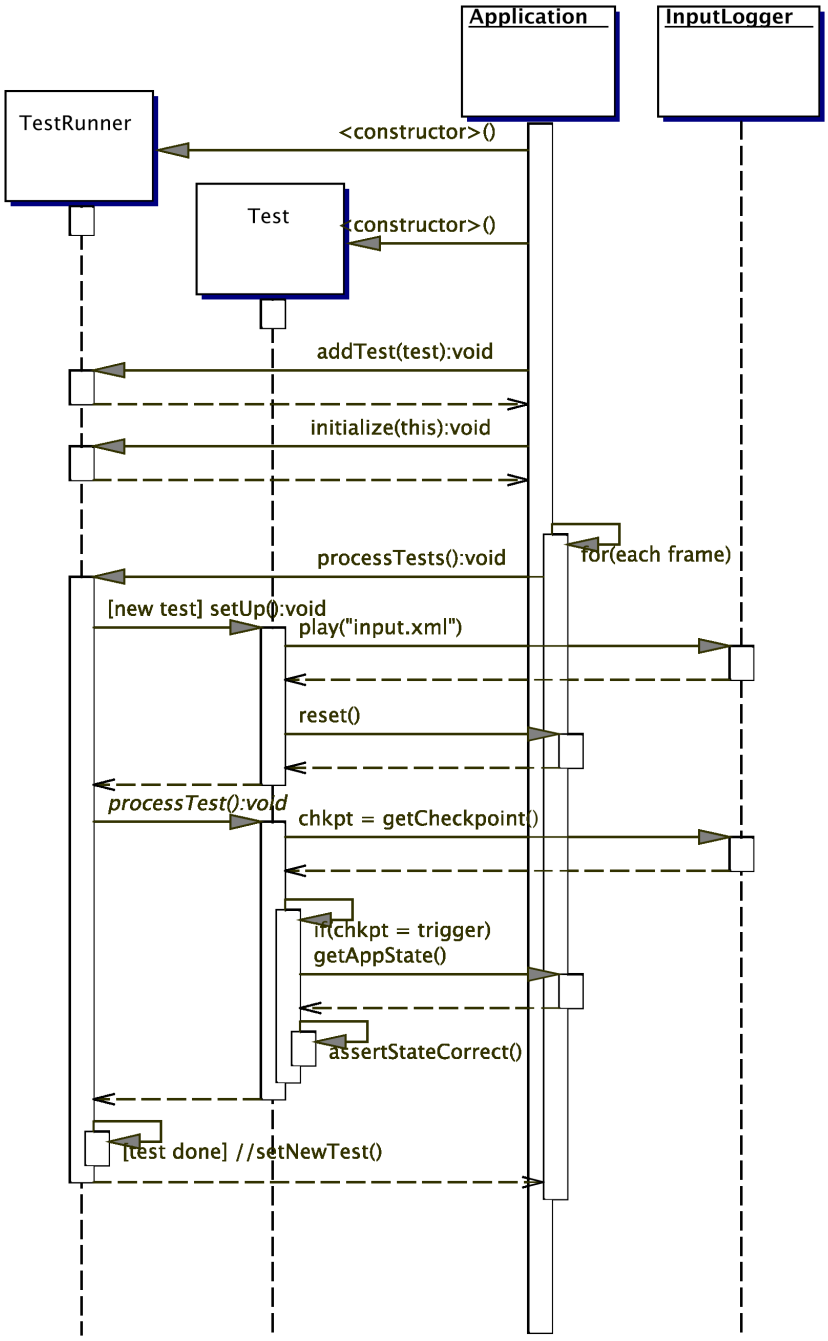


Figure 2.5: Test execution

Chapter 3

Automated Graphical Regression Testing for Virtual Environments

This chapter presents an analysis of the special requirements for testing user interfaces in virtual environments followed by an overview of the main contribution of our work: a testing process and an automated tool for regression testing of virtual environments.

While several well understood and widely accepted procedures for software testing in the GUI and regression testing area exists, these procedures either focuses specifically on desktop applications and other application presenting the users with a "standard" user interface, or at least makes no mention of the special considerations of virtual environments. In this project the existing procedures are analyzed with respect to virtual environments and a test procedure, based on the existing testing procedures, that is suited for virtual environments is found.

To help developers follow and utilize the advantages of standardized software testing development, we have developed different tools. This includes unit testing tools like CppUnit, test automation tools like Cruise Control and NAnt, and different capture-and-replay tools. These tools all play a vital role in creating and maintaining a strict software testing regime, but, as with test procedures, are either focused directly on traditional software applications, or does not support virtual environments. In this project test tools that can be used to support testing procedures for virtual environments will be analyzed and designed. The use of existing tools that can be utilized when testing applications that use virtual environments will be discussed and examples of use shown.

3.1 Goal and Intentions

The overall goal of this project is to investigate how software testing procedures and their associated tools can be used when working with applications that use virtual environments. The intention is to bridge the gab that exists between software testing methodologies in traditional software development in compari-

son with game and simulation development, thereby allowing developers of the latter to achieve the advantages of software testing: removal of bugs, including faster and earlier bug detection [Horn and Grønbæk, 2008].

The following sections will summarize the considerations that should be taken when developing an automated test tool for use in virtual environments. The goal is to create a list of steps that, when followed, leads to a practical testing process for applications using virtual environments. When the steps have been identified, each steps is analyzed, and a method for performing the test step is found.

3.2 GUI Testing Virtual Environments

Section 2.1.2 introduced the concept of test oracles, which is used for creating and validating test of the user interface in an application. The information on the types of test oracles presented was done with desktop application GUIs as reference point, but the information will in this project be considered to be valid for virtual environments as well. This means that in this project the two parts that make up a test oracle, the oracle information and the oracle procedure, can also be constructed for an user interface in a virtual environment.

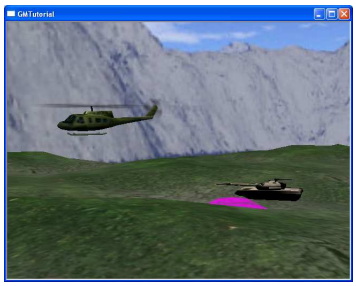
3.2.1 Oracle Information in Virtual Environments

For an application presenting a user interface of the type found in virtual environments, several artifacts can be used as oracle information. If the application includes some traditional GUI elements, like the GUI shown in figure 3.1, the oracle information could for example include the values of the input fields and the text labels. This is similar to the information suggested as oracle information

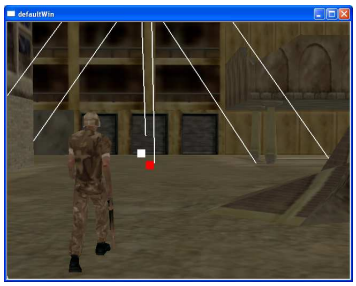
Figure 3.1: "Traditional" GUI embedded in a virtual environment

in traditional GUI testing, which implies that for the GUI element included in the graphical user interface, a standard oracle test procedure might be sufficient for testing the user interface.

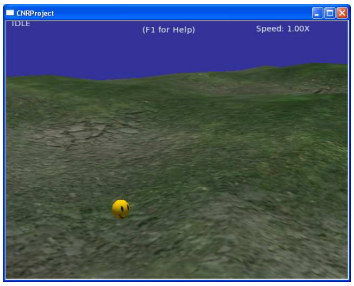
This project focuses on the elements of the user interface which are not related to the traditional GUI elements. This means that it is necessary to consider other elements of the interface which can be used as oracle information. The elements of particular interest in this project are all elements that are part of the virtual environment, and which are presented to the user through the user interface. Even though there is no real limit as to what these elements could include, for the sake of simplicity, a few specific elements in example applications have been selected which could be considered as oracle information. Figure 3.2(a) through 3.2(c) shows three different examples of user interfaces in virtual environments. All three example applications were developed using



(a) GMTutorial



(b) Walkingsoldier



(c) CNRProject

Figure 3.2: Three different Delta3D applications using virtual environments

Delta3D. In the application shown in figure 3.2(a) the tank shown in the middle right area is a moving actor controlled by the user of the application, while the helicopter is a statically placed object in the environment. In this application an example of oracle information available for testing could be the movement of the tank in the environment. Figure 3.2(b) shows an animated marine figure moving through an urban environment. The figure is automatically controlled the application AI (Artificial Intelligence), and the user is only left in control of choosing between a number of camera positions, relative to the marines position. The oracle information in this application is the heading of the marine and the type of movement he is doing (crouching, walking, running etc). The final figure, figure 3.2(c) shows a simple actor, a round smiley face, moving around in the virtual environment. The actors movement is controlled by the user of the application, and the user can place additional static objects, like boxes and

barrels, in the environment through the user interface. In this application the oracle information is the movements of the actor in the virtual environment, but also information contained in the application state, like the number of actors in the environment and their positions.

The three applications and the examples of oracle information mentioned for each application serves to illustrate the differences in the oracle information for virtual environments, seen in comparison with oracle information in desktop applications. These examples of oracle information for virtual environments are by no mean exhaustive, but are included only to point out the fundamental differences. A text label in a desktop application can be tested to hold a specific value or not, while the information whether the tank in the first example application has moved on the right path is a more vague parameter and harder to validate. Similarly, the movement and particular the animation of the marine in the example application, are harder to test, than simply examining if a given input field has focus in a desktop application. The third example is given to illustrate that there can be a combination of the test information in a desktop application and the interface of a virtual environment. In the third example, some of the oracle information can be examined as defined values in the application, that can be tested either true or false. The problem with the oracle information in this example, is that the application is controlled through an interface provided in the virtual environment, which it is harder to interact with for a test suite. To decide which data is really practical as oracle information, it is also necessary to look at the possibilities for creating different types of oracle procedures.

3.2.2 Oracle Procedures in Virtual Environments

In the previous section several examples of possible oracle information for applications using a virtual environment was suggested. The examples were based on sample applications developed using a specific technology, the Delta3D framework, and as such do not hold any general value as oracle information that could be used in all applications using virtual environments. Instead, the oracle information is selected based on the user interfaces of the selected applications, and the intended use of the applications. When selecting the oracle information an important consideration is whether there can be made an appropriate test procedure for the information. The oracle procedure is responsible for comparing the data in the oracle information with the values used for reference, and as such must have a logical way to do that comparison. For discrete values, like the values captured from input fields and text labels, it is easy to store expected values, and then compare the similarity of the values. For oracle information from virtual environments the development of a suitable oracle procedure is more challenging. To evaluate whether the tank in the example application is moving as expected, requires that the developer of the oracle procedure has considered what exactly constitutes “moving as expected”? It might be possible to create a tool that could compare exact translation and rotation of the tank at specific intervals, if that was required, with the oracle information. The problem with this approach is that the landscape of the application is created dynamically, and the height of the tank might not be exactly identical from execution to execution. Instead a more advanced comparison is needed. For exactly this reason the correctness testing procedure described in section 2.3.2 is often used. Here

a human replays a specific part of the simulation, and by inspecting the results visually evaluates the correctness. Similar examples and situations can be found for the other two applications. While it might be possible to create an exact oracle procedure for a given test, it requires a similar exact oracle information, or at least a very smart oracle procedure algorithm, which will take into account the special considerations that virtual environments present. Instead, a more "loose", but practical, evaluation is performed by a human tester.

3.2.3 Test Oracles in Virtual Environments

Based on the considerations on oracle information and oracle procedures in the previous sections, a concept for the creation of a test oracle that can be used for virtual environments is found.

Since there is virtually an unlimited number of possible oracle information data available in a virtual environment, and since that data is dependent upon the application content and use, the most practical way of attacking the problem, is to consider the oracle procedure first. It should be clear, based on the previous section, that the easiest and most flexible way to create an oracle procedure, is to use a human tester to compare the oracle information with the expected results. This is also the first of the four approaches to creating a test oracle that is specifically mentioned in section 2.1.2. Unfortunately, it is also the approach, that for a virtual environment application, is described as unfeasible in the long run. The second approach given in section 2.1.2, which also involves a human tester, supplements the human with a more automated system for interacting with the application. In this approach a human tester first creates test cases by interacting with the application, and asserts the correctness of the application. Later a capture-and-replay (CNR) tool repeats the user interaction sequence, and the relevant oracle information can be verified.

In this project, an approach similar to the described second type of test oracle is considered to be a feasible approach for testing virtual environments. By combining the flexibility of a human oracle procedure with the advantages of an automated approach the best of both solutions can be captured in one tool.

Based on the suggested steps for a GUI test procedure from section 2.1.4, which are summarized in table 3.1, the test oracle will be described in the following. While step one and six are listed as part of the GUI test procedure,

step#	name
1	Decide test coverage criteria.
2	Generate test input.
3	Generate test output.
4	Execute test cases.
5	Compare outputs.
6	Check test coverage criteria.

Table 3.1: Steps in software GUI testing

they will be considered out of the scope of the actual development of a test procedure for virtual environments. Step one and six, while of an analytical nature, can still be assisted with testing tools, will not be considered further in

this project. Step two through five are considered to be steps that involve the use of a test tool, and are therefore of interest in this project. When these steps are considered in relation to the previous assertions about using humans as test procedures it is clear that step two, three and five are where a human tester is useful, while step four is exactly the repetitive work that needs to be handled by an automated tool.

To summarize, when developing a test tool for virtual environments, the tool should accommodate the four steps above, in such a way that it helps a human tester to perform the steps that must be undertaken to assert correctness of the application under test.

3.3 Regression Testing Virtual Environments

In section 2.2.3 a number of suggested steps to follow when using a regression testing process were listed. The steps listed include a complete software change process, which means that it includes several steps that are concerned with work that must be undertaken previously to or after the actual testing. Table 3.2 shows the steps described in short form. Of these steps, step one and two

step#	name
1	modification request
2	software artifact modification
3	test case selection
4	test execution
5	failure identification
6	fault identification
7	fault mitigation

Table 3.2: Steps in software regression testing

are concerned with the software change process. Step six and seven are also concerned with actual software change, and as such not part of a testing procedure. This leaves step three, four and five, which in this project are considered to be the actual testing procedure. The steps listed in table 3.2 are not specific for any particular type of application, and there is no reason to assume that the procedure should be more suited for desktop applications than for virtual environment applications. It is worth noticing that according to the list, the actual creation of test cases are not as such a part of the software change process.

3.3.1 Testing Procedure Evaluation for VE's

The test tool that is being developed in this project should ideally support the three steps marked in table 3.2. The following list describes what is expected of a test tool in each of the three steps.

3. Selection - the test tool should help the tester with selecting the right test cases for a given execution of test system.

4. Execution - the test tool should help the tester with running the selected test cases.

5. Identification - after test execution the test tool should help the tester with identifying failed tests.

When considering how the developed test tool should help facilitate the selected steps, the particular requirements to such a test tool are based on the need to accommodate a testing procedure for virtual environments, as described in the previous section.

In this project, a test case for virtual environments would typically consist of a specific input sequence which the application should replay, information on when oracle information, in the form of screen shots should be captured, and finally the material the oracle procedure should compare the oracle information with. A test tool that helps with test case selection should therefore assists the developer in selecting the test case, by presenting an overview of available test cases, along with some information on what failure each test case actually tests for. When a test case is selected it should be added to a queue of test cases that should be run.

A test execution in this project, is when one of the developed applications is executed with the selected test case as "input". So for each execution the test tool should ensure that the application performs every selected test case.

The final step that the test tool should help with in the test process, is the identification of test failures. Since the test tool uses a human oracle procedure, it is the task of the test tool to prepare the oracle information in a way, that the oracle procedure can be done fast and precisely.

3.4 Procedure and Support Tools for Testing Virtual Environments

Based on the previous sections this section first summarizes the information on GUI and regression testing for virtual environments. A testing procedure specially create for virtual environments is then presented in a step by step list, with commentaries. Next, the tools we have developed to support the procedure, are described. Finally, a case study, showing the main functionality of the tool, is presented.

The previous sections analyzed and evaluated testing procedures used and/or recommended for GUI testing and regression testing, and presented the problems and challenges that this project have considered, before creating a similar procedure for virtual environments. Based on the evaluation of the GUI testing process, and the need for creating good oracle information, that can be treated by a oracle procedure in a practical way, it was decided that the best way to approach the problem of testing virtual environments was to evolve the human test process. It is clear that the obvious way of doing testing of virtual environments is by performing a manual test, where the developer or tester responsible for a particular area of a simulation or game, executes the application under test, and evaluates the correctness of the application. Not only is this one of the procedures already suggested for GUI testing, it is also the procedure that seems most obvious after analyzing oracle information and oracles procedures. The experiences collected from cooperation with developers in the simulation development industry [Horn and Grønbæk, 2008], show that this is in practice

also the testing procedure already used when performing correctness evaluation of development projects.

Based on these experiences a decision was made, stating that the testing procedure would be based on a human oracle procedure. Thereby, it becomes clear that the challenge that must be solved in this project, is the problem arising from the in-feasibility of performing such manual testing for every single change introduced. For each use case being tested a completely manual testing procedure involves: configuration, interaction and evaluation, where especially the interaction step can be time consuming. Another challenge with using a manual testing procedure, is that potential benefits from using regression testing becomes significantly harder to obtain, since a manual testing procedure cannot be relied on to be 100% identical from execution to execution. Additionally, one of the key concepts of regression testing is the execution of a high number of tests at a regular schedule, for example after each commit, and this makes a manual testing procedure even more in-feasible, since the amount of time used on testing would increase very fast along with the size of a project.

3.4.1 Testing Procedure

In this section the test procedures from the previous sections are combined in to a single procedure that can be followed when testing applications using virtual environments. The description of the procedure steps are created with virtual environments in mind, and in particular with consideration to the combination of a human test procedure and test automation tools. Table 3.3 shows the selected steps, their originating procedure and a short description. The six steps

Step#	Step name	Origin	Description
1	Generate test input	GUI test step	Capture test input for later replay.
2	Generate test output	GUI test step	Capture and store screen shots from the application.
3	Test case selection	Regression test step	List available test cases, and help with selection.
4	Test Case Execution	Both test procedures	Perform replay of the selected test cases.
5	Compare outputs	GUI test step	Show output is in sensible fashion.
6	Failure identification	Regression test step	Help identify failed tests

Table 3.3: Procedure test steps

are the procedure that this project will use when testing virtual environments, and for which support tools are developed. In the next sections, the rational behind each step is explained, and each step is described in more detail, along with the support tool used in the step.

Generate Test Input

When testing an application using a virtual environment as its user interface, the use of a capturer-and-replay tool is a practical way of allowing the test to be run often and with-out the risk of introducing human errors into an execution. To facilitate such a replay, it is necessary to first capture an execution of the application, and then store this data in an appropriate way. It is important to notice that when storing this data, it must be the inputs for the execution that are stored, and not the actual "execution", for example in the form of a captured video. This is due to the fact, that to perform the following steps in this testing procedure, it must be the application under test which is executed, and not just a captured representation of the application. If an application is modified, then it should be the modified version that runs with the test input, and tested for correctness. It would not make any sense to simply rerun the capturer of a previous version of the application.

In practice this means that the most low level form of input to the application will be captured and stored, where this input is considered to be keyboard and mouse inputs. That input will be stored in a format that allows replay of the input in an application. To support this activity the tool *VE Test Support (VETS)* has been developed. By integrating the VETS tool in a Delta3D application, and configuring the component appropriately, it is possible for a tester to execute the application and perform the necessary interaction with the application, and have the input to the application automatically stored.

Generate Test Output

The considerations about different test output types in the form of oracle information was discussed in the previous sections. While many different formats can be considered, and more than one type might be possible to use, the obvious choice is to use screen shots from the application. When a human tester evaluates the correctness, it is the visual aspect of the user interface that gets evaluated. Even if the test case involves checking for the correct sequence of a number of events or the position of a specific actor, these states get evaluated by the tester, by visually inspecting that the simulation is performing as expected. This will also be true in this project, where a human oracle procedure is used.

In this project this is interpreted so that it must be possible for a tester to chose and store screen shots from an application for visual inspection. This is done by including functionality into the tool for the input logger, which allows the tool to capture screen shots, and store these in a format "readable" by the programmer/tester of the application. By integrating the test output generator into the test input generator two features are implemented simultaneously: reference output generation and oracle information generation. When a tester is recording the input sequence as described in the previous section, the reference output are generated by invoking a screen shot capturer function bound to a keyboard key. Thus, when the tester identifies a scene or object that should be used for oracle information, the tester notifies the input logger application which takes and stores screen shots appropriately. When the test case is executed, at a later time as part of the test procedure, the input sequences that triggered the original screen shot captures are also replayed, just as the regular input sequence, and new screen shots are captured. These new screen shots are the

oracle information, and the tester can inspect the new screen shots against the reference screen shots. This functionality is implemented as part of the default functionality of the input logger.

The screen shot capturer process can be done in several ways, depending on the requirements to the test being performed. The simplest way is the method described above, which is also the default solution in the input logger tool. In addition, a more advanced image capturing tool has been designed, which can replace the role of the standard capturer tool. This advanced tool, the *Image Collector*, can take multiple screen shots at a number of angles and positions, thus allowing the tester to potentially inspect the oracle information in a more nuanced way. By designing the input logger as a framework, with a default implementation, this project aims at allowing a developer to utilize the tool and technique most appropriate at the time of testing.

Test Case Selection

In the test case selection procedure step, the tester manager of a software system should be provided with the ability to select relevant test cases that must be run for each test "session". This is of particular importance when testing virtual environments by using the tools developed in this project. Each test case runs in real time, and a large number of test cases lead to long test executions, which makes it practical to be able to perform only selected and relevant test cases, rather than a complete execution of all test cases.

In this project all test cases are kept and stored for use at a later stage, in a library of test cases, as recommended by [MSDN, 2008] and others. Since the test cases are associated with a specific development project, the test case library is stored together with the project artifacts, such as source code, build files, maps, etc. Figure 3.3 shows an example of how the test cases are organized

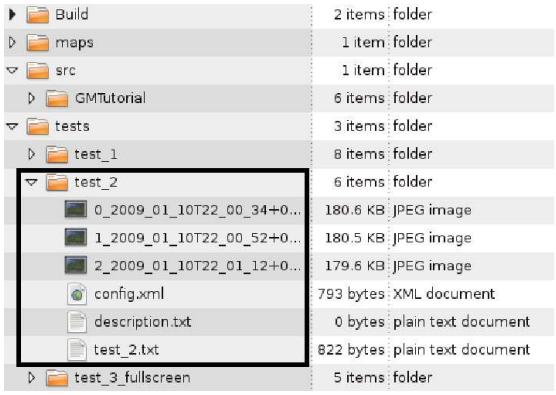


Figure 3.3: Test Case library structure

in sub-folders under a test folder for a project. Each subfolder is considered to be a complete test case, with images that are used for the reference material, a `config.xml` file to configure the Delta3D environment, a `description.txt` file which presents a short introduction to the test case, and finally, the input file captured by the VETS tool, here the `test_2.txt` file, which is used to provide input for the automated execution. When using the above folder structure,

it was found that is was possible to select relevant test cases for execution in a simple and organized fashion, without designing and implementing a more advanced tool. Section 6.5.1 describes how the libraries are combined with the test execution tool to perform selected test cases.

Test Case Execution

The execution of test cases is a central step in the testing procedure, where the test input is combined with the application under test, through a replay tool, and the test output is generated. There are several considerations when designing an execution tool. In section 2.3 a presentation of a system for testing virtual environments was reviewed, and a small design example was given. In this example a `TestRunner` class was designed and implemented along with a special `TestCase` class, which was responsible for inquiring the state of the application at specific points in the execution. While a similar system could be devised for this project, the testing procedure that is being developed in this project is significantly different than the one presented by [Bierbaum et al., 2003].

In this project, the oracle information is collected in form of screen shots, and the oracle procedure is performed by a human, as opposed to the more low-level application states, which are tested using automated test oracles in the design presented. This has the effect, that, a more simple and general purpose test execution strategy can be inferred. The requirements to the test execution, is basically that it must be able to run the application, including the components designed in this project. To simplify test execution, it was decided to use standard industrial software automation tools, specifically Cruise Control .NET [2] and NAnt [3]. Cruise Control .NET, allows setup of an automated build cycle, while NAnt allows advanced scripting and application execution. Both tools were presented and discussed in relation to software testing and test automation in [Horn and Grønbæk, 2008].

Compare Outputs

When utilizing a human oracle procedure in a project such as is this, it is important to help support the oracle procedure flow. The main risk when using a manual, rather than a completely automated, testing procedure, is that the manual work grows to an amount that makes it impractical or even unfeasible to perform. To alleviate this situation the manual oracle procedure must be supported.

In this project the manual oracle procedure consists of evaluating the correctness of the application by comparing images captured during application execution. To facilitate this, a *Test Reporter* tool that allows the tester to view the images he is comparing in a practical way, has been developed. After each test case execution, the tool can be invoked and automatically generate a HTML format report, where the images are presented side by side. This allows the tester to quickly and efficiently evaluate if the correctness of the application can be confirmed. In addition to the images, the report also presents a description of the test case, so that the developer can perform an evaluation of the images. The output comparison step is done in parallel with the failure identification step.

Failure Identification

Part of the oracle procedure is not only to identify failed test cases, but also to, if possible, help identifying the failure itself. In a standard testing procedure, using an automated tools, failure identification involves pinpointing the exact location of the test failure, for example a specific field holding a wrong value, or a internal variable not being set. In a testing procedure using image comparison, the failure identification is more uncertain, since it is harder to detect miniscule differences in images. When the test case involves many images it becomes increasingly unrealistic to closely inspect every image.

To help solve the above problem with failure identification, the *Image Analyzer* component has been developed in this project. The tool serves a dual purpose, first helping testers spotting the exact location of differences in two images, secondly, by reporting a collection of image metrics that can be used to identify images that are not identical. The Image Analysis tool is used by the Test Reporter tool, and its output are shown in the image comparison report along with the reference images and oracle information.

3.4.2 Support Tools

To support the testing procedure developed for in this project, a number of test tools that has been developed. The tools developed are summarized in the following list:

- VE Test Support : The VE test support tool has been developed for Delta3D Game Manager, and allows for recording and replay of simulations. In addition, a default implementation includes a simple image capturer tool, which can be replaced by the more advanced Image collector tool if necessary.

- Image Collector: Responsible of collecting images in a VE based on inputs made by the tester. The inputs correspond to the test selection step. The collected images are stored in order to make the comparison with reference images possible.

- Image Analyzer: Responsible of analyzing the collected images using image processing algorithms. The degree of similarity between collected images and stored reference images is calculated. The result of the image processing algorithms corresponds to the execution step.

- Test Reporter: Responsible of presenting the result of the regression test to the tester. The result of the test execution determines whether the tester evaluates the test as passed or failed. The test result corresponds to the identification step.

To make the execution of regression testing in virtual environments semi-automatic, each of the above components are controlled using NAnt and Cruise Control.

VE Test Support

The VE Test Support tool is designed and implemented as a Delta3D component, which can be added to existing or new Delta3D applications using the

Game Manager framework. As the name applies, the primary role of the VETS tool is to support a human tester in performing test procedures, by automating test steps where possible.

Image Collector

A component in the test tool developed in this project is the image collector, which aids the tester in collecting scene images in a virtual environment. The collected images are used at later stages in the testing procedure to determine whether or not two graphical scenes are identical.

Ideally, a fully automated regression testing tool for virtual environments which does not requires human intervention would be preferred. Unfortunately, such tools do not seem to exist. Instead, semi-automated techniques are used. The strategy used in this project is to capture scene frames and store these as images and compare them with reference images of the same scene frames. The simplest form of this approach requires human intervention. Here testers study the captured images and determine if the images match corresponding reference images. This technique relies entirely on the judgement of the tester studying the images. By making the matching decision a matter of human interpretation does impose certain problems, because different testers might estimate differently.

Because of this, the image collector is only responsible of collecting image material. The task of processing the images is delegated to other components in the regression test tool.

Image Analyzer

Based on the images collected during execution of a virtual environment application, the image analyzer performs an analysis of the scene images, where the degree of similarity is calculated.

The strategy used by the image analyzer is based on techniques from computer vision and digital image processing. Various images are collected at runtime during the simulation or while running the game. The collected images are later processed by the image analyzer to establish whether scenes have been altered compared to the stored reference images. The result is made available to the tester in a status report, containing pass/fail entries for each comparison performed.

Determining if two scenes from a simulation or a game, when taken at the same stages but at different times, differ or are identical by capturing and comparing their 2D images can be a complicated affair - especially if the scenes are from a 3D simulator or a 3D game. In this case, details are obviously lost, since a 3D scene contains more information than its projected 2D image. A naive approach is to iterate through the collected images, and for each image perform a pixel by pixel comparison with the corresponding reference image. However, there are several problems with this approach. First, if one of the scenes suffer from z-buffer flickering, the corresponding image inherits this in form of pixel errors or more precisely the colors of certain pixels vary from the colors originally in the texture. Secondly, in pixel by pixel comparison every pixel is weighted the same. This means that distinction between certain regions in the image becomes hard, because no information is kept about which object

a pixel belongs to. Therefore, the result of a scene comparison performed using the image analyzer is a number indicating the percentage of similarity between the involved scenes. Additionally, the image analyzer is able to subtract the scene images and present the result as a new image showing the differences between the scenes. Visually showing potential dissimilarities has the advantage that the tester quickly can spot the inaccuracies and react accordingly.

Test Reporter

The test reporter presents the result of the entire regression test. For virtual environments the result is in form of a status report showing the results of each of the selected tests. This includes visually showing both the collected scene images and their referenced counterparts together with the calculated similarity estimates. Additionally, the visual dissimilarities between selected scene images are presented to the tester as difference images making it easier for the tester to decide whether the test should pass or fail.

3.5 Architecture

Figure 3.4 shows a component diagram of the regression test tool developed in this project. In the following a description of each of the components is given, including how interaction between the components is handled.

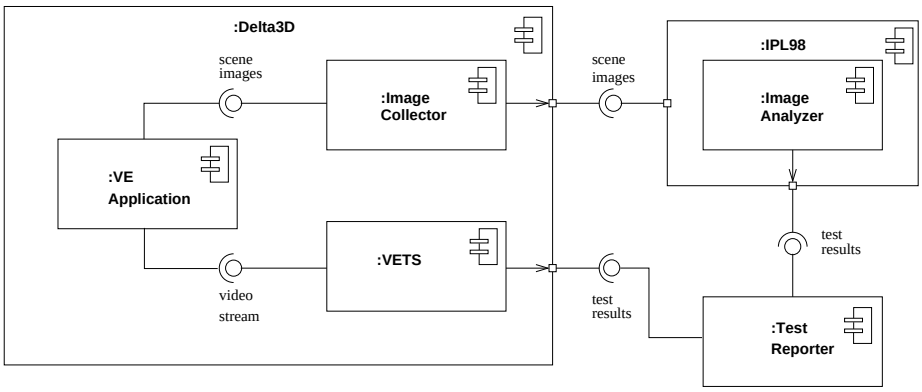


Figure 3.4: Component diagram showing the architecture of the regression test tool.

3.5.1 Delta3D Virtual Environment Application

The VE application component represents the application under regression test. This component could be any graphical application developed using the Delta3D framework engine. Therefore, a couple of graphical demo applications have been implemented in order to have some Delta3D applications to experiment with. The VE application corresponds to any of the demo applications mentioned above. The VE application component interacts with the image collector component. The interaction takes place programmatically in the sense that the

image collector component provides an extension to the existing Delta3D class library.

3.5.2 Image Collector

The image collector implements functionality to capture multiple scene views in a virtual environment application at run-time. The image collector is assembled in a framework with low coupling to the existing Delta3D class library.

3.5.3 GM VETS Tool

The VETS tool is a framework increment designed to capture and log keyboard events from a Delta3D application. The usefulness of these features, in the context of regression testing, is to be able to perform after action reviews or playbacks of virtual environment applications. In regard to regression testing and the ability to automate testing of virtual environment applications, the VETS tool makes it possible to compare application sessions with stored reference. The VETS tool is implemented using the existing Delta3D class library, which makes it a candidate for inclusion into future releases of the engine.

3.5.4 Image Analyzer

The image analyzer component works independently of Delta3D, and could be regarded as a self contained collection of classes. The purpose of the image analyzer, is to compare the scene images, collected during a virtual environment application, with stored references. The image analyzer performs these comparisons using image processing algorithms, which estimate the degree of similarity between the scene images. Additionally, the image analyzer component has functionality to visualize the difference between images.

3.5.5 Test Reporter

The test reporter is a Java application, which is responsible for presenting the result of a regression test case execution. This is done by generating a HTML report which presents both the reference screen shots that make up the test case, the oracle information captured and the analysis results from the Image Analyzer component.

3.5.6 Support Tools

The previous sections briefly introduced the behavior and responsibility of each of the components contained in the tool. The following section describes how the components collaborate in order to realize a semi-automatic regression test tool.

In order to make the regression testing tool as autonomous as possible an automated build process has been used, where all source code has been kept at a subversion server. Additionally, Cruise Control has been used as build server in order to control the build process. The build process has been directed by configuration scripts to ensure automatic check out of the most recent version of the application source code. Scheduled nightly builds together with NAnt scripts have been used to build, compile, and run the applications under regression test.

3.6 Case Study

The preceding sections have given an overall description of the problems involved in developing a tool for testing applications containing 3D graphical virtual environments. Additionally, a general description of the components included in the implemented prototype has been presented. Before presenting a more exhaustive description of the developed prototype, a case study description is given.

In the following the WalkingSoldier application, which has been developed to provide an experimental VE application, is used as a case study. The WalkingSoldier application is described in detail later in this report. Therefore, the following only gives an overall description of the application. The WalkingSoldier application is a Delta3D application, containing an animated soldier character. The soldier navigates through an urban city environment, containing various static objects like buildings, rocks, trees, bushes, etc. - all having textures.

Suppose the soldier should carry a weapon at specific stages during the simulation, and a programmer wants to assert that the soldier in fact is carrying a weapon at these stages. The programmer inserts multiple scene views, using the image collector module, at the critical stages in the source code responsible of visualizing the soldier. The programmer runs the simulation and asserts, by studying the collected bitmap images together with the calculated similarity estimates, that the soldier carries a weapon at the correct stages. The change is committed to the repository holding the application source code. Suppose another programmer, working on the application logic in the WalkingSoldier application, unintentionally modifies the behavior of the soldier, so that he no longer carries a weapon at the specific stages in the simulation. The programmer compiles, builds and runs the application locally and determines that everything is in order. Therefore, the change is committed to the source code repository. The build server, running nightly builds, checks out the most recent version of the source code and subsequently compiles, builds and runs the application. At run-time, unit tests capture multiple scene views of preselected locations within the virtual world and store these views as bitmap images. A script, running on the server, executes the image analyzer module which compares the collected images with stored reference images. The image analyzer module calculates the degree of similarity between the collected images and their previously verified counterparts. Upon termination of the unit tests, a report showing the collected images, the stored reference images, and the difference images is generated together with the similarity estimates. The report is published on the build server as a html-document. The next day, the programmer responsible of the change, looks through the report and realizes that his change caused conflicts.

Chapter 4

A Testing Framework for Delta3D

This chapter presents the central framework extension we have implemented to enable regression testing in Delta3D. First, we provide an introduction to the use of object-oriented frameworks in software development, including motivation, a design guide and information on how framework instantiation should be done. The introduction of frameworks is presented with the intention of understanding the basic properties and motivations behind framework design, and then in the following sections evaluating a couple of central elements in the Delta3D framework seen in this light. The goal of this evaluation is first to acquire knowledge on the internal structures of the existing Delta3D framework, and investigating where, if any, would be the optimal position to expand the framework with new functionality. Secondly, by examining the Delta3D framework, the existing build-in capabilities that could be used for implementing a tool for software testing is considered.

The first section provides a general high-level introduction to frameworks. All mentions of frameworks in this section refers to object-oriented frameworks. The goal of this section is to provide background information on frameworks which will then be put in to a Delta3D perspective in the following section. The second section introduces a selected central part of the Delta3D framework, the Game Manager. While the Delta3D framework consists of many classes and namespaces, the Game Manager can be considered to be a complete subset of the larger framework, and provide functionality to developers by fulfilling a defined role in the overall framework. The concept of roles are introduced in the following section. The third section evaluates a particular instantiation of the Game Manager which provides a Delta3D application with advanced record and playback functionality.

4.1 Framework Design and Implementation

Frameworks are used in the software community to allow companies to capture the commonalities between applications for the domain they operate in. [Bosch et al.,] provides a loose definition of a framework as:

> A framework consists of a large structure that can be reused as a whole for the construction of a new system.

[Bosch et al.,] writes that in addition to the intuitive appeal of frameworks, frameworks help capturing commonalities in applications and allow developers to reuse existing code and thus reduce development time and decrease maintenance cost.

4.1.1 Framework Design

[Van Gurp and Bosch, 2000] provides insight into a conceptual model for designing frameworks and guidelines for building good frameworks. According to [Van Gurp and Bosch, 2000] a framework is defined as:

> a partial design and implementation for an application in a given domain.

This definition can be seen to mean that a framework is an incomplete system which is tailored for general use, and which must be adapted for use in particular applications. [Bosch et al.,] writes that most authors agree that a framework is a reusable software architecture comprising both design and code. A more formal definition is:

> A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

The particular adaption of a framework to use in a specific application is called *framework instantiation*. Frameworks can be grouped into three categories according to their use:

- Application frameworks which provide the complete functionality to create a full application. An example of such a framework could be the Java Foundation Classes which provides complete user interfaces for Java programs. Other examples include the similar Microsoft Foundation Classes.
- Domain frameworks are typically used for developing programs for a specific domain, like financial software etc. Since such applications often have similarities a framework can reduce the development time and heighten the quality of the programs.
- Support frameworks are used to simplify development in very specific areas, like memory management, and thus increasing productivity.

[Van Gurp and Bosch, 2000] lists two typical problems which arise in the use of domain frameworks: Composition problems and evolution problems. Composition problems arise when more than one framework is used, which can cause problems when specific frameworks try to control parts of an application already controlled by another framework. Evolutionary problems arise when, just as applications, frameworks are continually evolving in response to user request and new requirements. This might change a framework API, which might require changes to applications using the framework. The development and use of frameworks generally suffer from many of the same problems seen in classical program development. Frameworks grow to large and complex entities over

time, in response to new requirements. Restructuring of code and addition of code fixes increases complexity and the framework may eventually need a complete rewrite to enable further development. As in application development it is essential to design frameworks that are prepared for future changes.

4.1.2 Organization of Frameworks

Most frameworks start out by creating a few classes implementing some needed functionality, which are then generalized into a number of interfaces for use in other applications. This small framework is then increased in size and functionality over time as components are added to meet feature requests.

[Van Gurp and Bosch, 2000] distinguish between two different types of code reuse: blackbox and whitebox reuse, which can then be used to classify two types of frameworks. [Bosch et al.,] writes that a framework usually evolves from a whitebox framework into a blackbox framework. Blackbox reuse is when an existing implementation is reused by relying on the implementation's interfaces and specifications. In a blackbox framework this means that the framework provides existing components that can be used by configuring the components. This provides an easy to use framework, where the developer does not need to know of the internals in the framework, but also reduces the flexibility of the framework to pre-defined areas of development. Whitebox reuse is when reuse is done through the interface of the code, and by studying the actual implementation. In a whitebox framework this translates to a system where all use is done through inheritance, and where there is no ready-to-use implementation. The implementation must be done using the knowledge gained by studying the internals of the framework. This provides maximum flexibility for the developer using the framework. [Van Gurp and Bosch, 2000] recommend a combination of a whitebox and blackbox to increase usability of a framework. The framework should have a whitebox layer with interfaces and some abstract classes, which provide the architecture of the framework. Following that there should be a blackbox layer with concrete classes and components that use the whitebox layer and can be used immediately. This way the framework is both flexible and ready for use. Figure 4.1 shows the relationship between the white and black

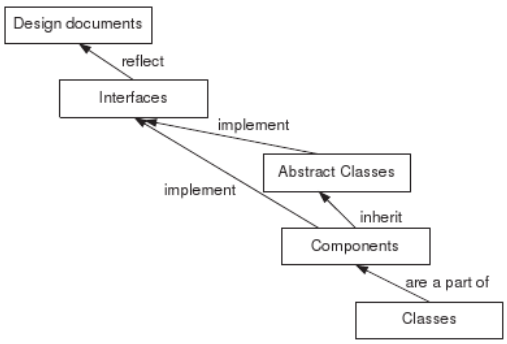


Figure 4.1: Whitebox and blackbox relationship

layer. Here the interfaces and the abstract classes are the whitebox layer of the

framework. The components and the (concrete) classes are the blackbox layer of the framework. [Van Gurp and Bosch, 2000] distinguishes between components and classes in such a way that the only difference is that components are classes that exposes their API through the interfaces in the framework, while the classes are part of the underlying implementation, and do not expose them self to developers using the framework.

4.1.3 Conceptual Model

[Van Gurp and Bosch, 2000] presents a conceptual model of a framework, as shown in figure 4.1 where blackbox frameworks are made up of individual components. Components can be made up of one to a few classes, in which case the component is called an atomic component, or it can be made up of several components connected with glue-code classes, in which case it is called a composed component. The highest level a composed component can reach is a complete application, with UI, application start/stop features and other functionality.

[Bosch et al.,] separates a framework into core framework design, framework internal increment and application specific increment. The core framework is a collection of both abstract and concrete classes, where the concrete classes are invisible to the framework user. An abstract class that must be implemented by the framework user is called *hotspot* by [Bosch et al.,], which is meant to separate the class from other abstract classes that might be internal classes in the framework and thus invisible to the framework user. The framework internal increments consists of additional classes, constructed to increase the usability of the framework. [Bosch et al.,] calls these internal increments to distinguish these classes from more general classes belonging to class libraries. These internal increments are for example used to realize abstract concepts that are often used in the application domain. An example could be to include several concrete subclasses of an abstract class, like `Device`, in the framework, to capture reuse of commonly used devices.

[Van Gurp and Bosch, 2000] introduces the concept of roles in relation to components in a framework. A role represents a set of functionality which must be provided to an application. This role can be fulfilled by using a subset of functionality provided by one or more components, and combining these. This means that a component can provide a different API to different clients, depending on the particular role the client is expecting. Each role a framework is able to fulfill is represented through a single interface in the whitebox framework, where the interface may be composed by several components. This provides flexibility and enables programmers to use the framework without making assumptions on the behavior of the components.

4.1.4 Framework Instantiation

When using an framework that is constructed as shown in figure 4.1 it is necessary to perform framework instantiation to adapt the framework to the application being developed. The instantiation can take place in one of three ways, according to the state of the application and the framework. If the framework provides exactly the components required for development in its blackbox layer, only configuration of the components and some glue code connecting the application and framework is necessary. If the components do not cover the

required functionality, it is necessary to implement application specific components. This is done using the interfaces and abstract classes of the whitebox layer, and inclusion of the application specific components in the framework should be considered. Finally, it might be necessary to implement additional application classes to provide the needed functionality. If this is the case, and this is done several times, a new framework should be considered.

4.1.5 Framework Guidelines

[Van Gurp and Bosch, 2000] list a number of recommendations in a short guideline to create improved frameworks, based on the conceptual model presented.

1. The interface of a component should be separate from its implementation.
2. Interfaces should be role oriented.
3. Role inheritance should be used to combine different roles.
4. Prefer loose coupling over delegation.
5. Prefer delegation over inheritance.
6. Use small components.

Apart from the specific advise, [Van Gurp and Bosch, 2000] provide several general recommendations: use standard technology to avoid the *not invented here* problem, use automate configuration tools to configure the framework since there will be much configuration with the mentioned model, and finally, use automated documentation tools.

4.2 Delta3D Framework Architecture

Delta3D is an open source game and simulation engine developed and maintained by the MOVES Institute Naval Postgraduate School in California USA. Delta3D was developed due to continuous requests by the U.S. military for new simulation software applied in various fields maintained by the Department of Defense (DoD). By investigating previously developed simulators, researchers at MOVES institute identified a number of commonalities in the underlying structure supporting the different simulators, and decided to extract these into a separate entity. The result of this work, is a framework containing a number of components often used in the development of modeling and simulation software [McDowell et al., 2006] and [McDowell et al., 2005].

Although, the motivation behind Delta3D, was to provide developers of modeling and simulation software in the context of military training systems, with a library containing an application programming interface (API), the use of Delta3D as a graphics engine has been applied in the game industry as well.

4.2.1 Libraries In Delta3D

Delta3 is distributed as a complete visual modeling engine, not just a game engine. This means that apart from the game engine parts Delta3D also supports and makes different technologies available to people using the engine. The engine

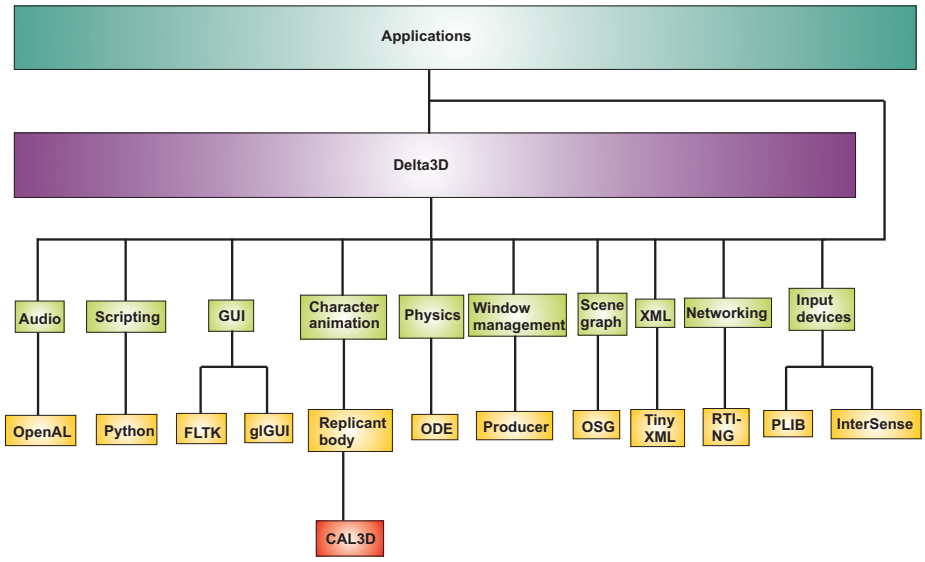


Figure 4.2: Library hierarchy in Delta3D.

makes use of several libraries as seen in figure 4.2. Some key components found in the engine are: Support for audio, a character animation library, physics engine, rendering engine and support for scripting and AI functionality. A detailed installation and configuration guide explaining the process of compiling and building Delta3D in visual studio is found in appendix A.

Audio

Audio in Delta3D is handled through the Open Audio Library, which is a software interface to the audio hardware. It intentionally resembles the OpenGL API in coding style and conventions and uses a syntax resembling that of OpenGL where applicable. The interface consists of a number of functions that allow a programmer to specify the objects and operations in producing high-quality audio output, specifically multichannel output of 3D arrangements of sound sources around a listener. To a programmer, OpenAL is a set of commands that allow the specification of sound sources and a listener in three dimensions, combined with commands that control how these sound sources are rendered into the output buffer.

Character Animation

Delta3D uses the Character Animation Library 3D (Cal3D) to animate characters. Cal3D is a skeletal based 3D character animation library written i C++. Cal3D makes it possible to use exporters, which are plug-ins for the most popular (both open source and proprietary) 3D animation tools. This means that level artists, or content creators, and animators can use their preferred modeling tools to create characters, animations, and textures, and then output them into a format Cal3D understands. The Cal3D C++ library in addition to loading exported files, build characters, run animations, and access the data necessary

to render them with 3D graphics. Cal3D can perform animation blending, which allows multiple animations to be executed at the same time with Cal3D blending them together smoothly. This effect allows characters to transition smoothly between different animations, such as walking and running.

Physics

Physics in Delta3D is simulated by the Open Dynamics Engine (ODE) library. ODE is a high performance library for simulating rigid body dynamics. It is fully featured, stable, mature, and platform independent with an easy to use C++ API. It is currently used in several computer games, 3D authoring tools, and simulation tools. ODE can realistically model several devices/physical phenomena, such as joints, springs, damping devices (e.g. shock absorbers),friction, gears, motors, and collisions. Advanced rigid body mechanics can be build from these simulations, providing realistic behavior of objects in the game world. ODE uses low-order integration and constraint-based actuators to reduce the amount of time tuning that a developer needs to use to create this realistic behavior. It is particularly useful for simulating vehicles, objects in virtual reality environments, and virtual creatures.

Rendering

For rendering, Delta3D uses OpenSceneGraph (OSG). OSG is an open source 3D graphics toolkit, used in fields such as visual simulation, games, virtual reality, scientific visualization, and modeling. It is written in Standard C++ and uses OpenGL as its underlying rendering API. OSG is explained in more detail later.

Scripting

The scripting language is one of the most critical factors in allowing advanced behaviors to be added to a simulator with a minimum of programming in the source language - that is languages like C++, C#, Java etc. Therefore a large number of engines supports scripting as a tool for the programmers to extend their applications. Delta3D uses the Python scripting language, which is a portable interpreted object oriented programming language, which has been in development since 1990. Python uses dynamic types, which makes it an elegant scripting language without making its syntax oversimplified. Although Python has a relative small number of in-built high level data types, it allows programmers to extend Python by adding new modules implemented in a compiled language such as C or C++. Such extension modules can define new functions and variables as well as new object types. Python includes classes, a full set of string operations, automated memory management, garbage collection, and exception handling [van Rossum, 1999b] and [van Rossum, 1999a].

Additional Functionality

Before the development of Delta3D was initiated, a number of functional requirements to be included in the framework engine were specified. Some of these requirements could be fulfilled by including existing open source projects into the framework, where each project encapsulated some functionality useful

in the development of visual simulators. Although, Delta3D make use of many existing open source projects, a number of functionalities were still missing to complete the list of required features to be included in the engine. Even though some of these features were available in form of proprietary modules from commercial software vendors, these could not be included, because Delta3D was decided to be licensed as open source. Therefore sponsors have funded the development of additional features, either at the Naval Postgraduate School or at other companies. The features includes:

1. **Graphical level editor:** Will greatly aid in placing objects at desired locations in the rendered scene and provide visual manipulation of these objects.

2. **Advanced terrain/vegetation rendering methods**: Provides algorithms that generate synthetic terrain by selecting objects such as rocks, shrubs, trees etc., and place these randomly in the scene, making the generation of a geological typical terrain an easier task.

3. **Advanced environmental features:** Provides the engine with advanced features like varying lighting conditions by changing time of day, moving clouds, effect of wind etc.

4. **Particle system editor:** Allows the engine to generate special particle effects such as dust, fog, rain, and smoke - effects that greatly add to the realism of the simulator. Alternatively, such effects can be imported from other simulators.

5. **Record and playback capability:** Provides the ability to record and play back certain stages of the simulation. This is an important and valuable feature in military simulations, making it possible to do an action review that aids in correcting mistakes by users of the simulation.

6. **3D model viewer:** Forms an important part of any system, which would aid in expansion of the assets library by adding new models.

4.2.2 AI In Delta3D

According to [Darken et al., 2007] Delta3D initially lacked support for creating reliable and reasonable behavior for non-controllable characters. Therefore it was decided to add artificial intelligence (AI) to the engine. Delta3D offers three basic facilities to support the development of AI. These include a finite state machine class, traditional waypoint based navigation and path calculation, and the ability to program AI in a high level scripting language.

Finite State Machines

Finite State Machines (FSM's) are often used as a modelling tool in theoretical computer science, but have found a more applied use in game engines. Here FSM's have been adopted for creating AI. FSM's are composed of a finite set of states and a set of transitions amongst them. There is also a finite set of input symbols ("events" in Delta3D) that cause the state to change. Each transition is labeled with an input symbol. At any moment, the AI is said to be "in" one

of the states. When a new input symbol is sent to the FSM for processing, the set of transitions from this state to any other state, is checked to see if any of these states are labeled with the new symbol. If one of the transitions is labeled with the new symbol, the current state is changed to be the one indicated by the transition.

Path Planning

A navigational infrastructure is necessary to allow non-player-controllable (NPC) characters to move around in a game terrain without colliding with walls and other obstacles. The navigation infrastructure implemented in Delta3D is based on waypoints. That is mathematical points specified by three-dimensional coordinates. Waypoint can be used to combine intermediate destinations to a longer path. The waypoints constitute the nodes of a navigation graph. For each waypoint, a directed edge is added to the graph for each other waypoint that can be moved to along a straight line.

Perceptual Modeling

According to the authors of Delta3D, many games and military simulators lack support for determining battlefield visibility. A simple approach in visibility modelling is to use Line-of-Sight (LOS). LOS trace from the eye point of the observing character to the top of the target to make a visibility determination. If the LOS does not intersect any polygons, the character is considered to be aware of the target. This is a drastic oversimplification, because there are instances where LOS visibility is too generous or too conservative. The result is AI that sometimes engages targets that are invisible to a human player and sometimes fails to engage obvious visible targets. For a more detailed discussion about LOS see [Darken, 2004].

AI Scripting

Delta3D supports AI scripting in Python, which makes it possible for users to add partial or complete programs. Not every game engine provides a scripting language. The arguments for doing so are that scripting languages are typically easier to learn and faster to use, at least for small programs. This makes it possible for people with a much broader range of backgrounds to modify a simulation. Besides that, less time is often required to produce a given feature if a scripting languages is used instead of a traditional programming language.

4.2.3 Translation and Rotation in Delta3D

Almost all realistic 3D applications of a certain size, whether they are games or simulators, are composed of various objects placed in a virtual environment. Often these objects are created in external modelling software as separate 3D models, which later are imported into a 3D engine where they are rendered. Additionally, the engine handles the entire scene setup involving arranging the models, placing the cameras, adjusting the lightning conditions, and organizing the entities contained in the scene. After this initialization phase the scene can be displayed on a screen or an other visualization device using perspective projection. This technique results in a series of "still images" showing the

scene from different viewpoints. Due to the limited size of the displaying device and the extension of the virtual environment these images cannot contain the entire scene. Therefore, most game engines provide features for translating and rotating the camera and thereby controlling the viewpoint of the scene.

The viewpoint of a Delta3D scene is controlled by the position and orientation of the camera. A significant part of the software created in this project makes use of camera transformations. Especially, the module responsible of collecting scene images relies extensively on Delta3D's support for performing camera transformations. Unfortunately, the documentation totally lacks information on how Delta3D handles camera transformations. Therefore it was decided to collect this information by studying the relevant source code files in Delta3D. The result of this is a collection of equations describing how Delta3D handles camera transformations, which are presented in appendix B.

4.2.4 Delta3D Framework Evaluation

After having described the principles and recommendations in good framework design and presented some of the functionalities included in Delta3D, the following paragraph focuses on the evaluation of Delta3D in regard to how we have used the framework in this thesis.

We have used Delta3D in two ways. First, the framework has been used to develop virtual environment applications, like the WalikingSoldier application and the GMTutorial. Here the class library has been used in much the same way as class libraries are used in traditional software development, where the framework supplies relevant classes and methods to the programmer as black-box components. Used in this way, software development in Delta3D has been rather uncomplicated, although a virtual environment application, developed in Delta3D, must follow a general pattern where certain methods must be overridden and some classes are mandatory to subclass. Using this skeleton was problematic in the beginning due to the relative weak documentation included in Delta3D. Unfortunately, the framework does not provide a comprehensive document describing the API and how it is used in application development. Instead, developers are referred to look directly in the source code, which also lacks documentation for the majority of the classes and methods included in the API. Alternatively, the Delta3D user community[1] offers great support, especially the maintainers of the framework have provided valuable feedback to our questions.

Secondly, the framework has been used as a foundation for implementing our own components. Here, Delta3D has been used as a whitebox framework, in the sense that the graphical regression test components have interfaces to the Delta3D class library. Although, we have only used a small subset of the classes included in Delta3D, the classes that have been used, have all been encapsulated entities with restricted responsibility. This has made the implementation of the framework extensions relatively uncomplicated, since we have extended the class library through existing Delta3D interfaces. The framework components developed in this thesis have been integrated with Delta3D through a combination of composition and inheritance.

[1] www.delta3d.org

4.3 Delta3D Game Manager Architecture

This section describes the architecture behind and the motivation for creating the Delta3D Game Manager, along with the central components used by the Game Manager. The purpose of the section is to give an introduction to the Game Manager architecture, and provide a background knowledge in Delta3D architecture, which will be used to implement Delta3D components for the test tool being designed and implemented in this project. The architecture of the Game Manager is evaluated based on the general design information provided in the previous section. This evaluation is performed both to understand the internal structure of the Game Manager, and in particular to ensure that the designs and implementations created in this project can be easily used and reused in future version of Delta3D. By understanding the formal model outlined in the previous section and Delta3D's framework architecture, it should be possible to create high quality code for the framework.

4.3.1 Background and Motivation

The Delta3D Game Manager (GM) is a new architecture introduced in Delta3D around 2005 with the stated intention of alleviating several problems associated with software engineering in modern game development. Just as software engineering in general has evolved rapidly in the past one or two decades, the field of game development has also changed. Initially game development was much centered around development of algorithmic content, like path finding and non-player-character AI, but today the main development effort is done in more structural areas. In general it could be said that the development effort have shifted from implementation toward design areas when working on modern game projects. Complexity in game development has also increased with the size of development areas, which today includes for example game logic, engine code, textures, models and game content. The focus in modern game development is on the creation of game content, rather than algorithms and game logic. To facilitate this shift in development most projects today use high level game engines like Delta3D, which allows programmers to focus on design rather than implementation. Even with the use of game engines it is often still a major development effort to create application infrastructure, manage game elements, handle objects and develop communication between game entities.

The motivation for creating the Delta3D GM is to alleviate exactly the mentioned problems with managing interactions and connections between game elements like game actors and game components. The GM framework allows developers to concentrate on content creation and level design. In addition, the introduction of the GM aims at increasing the ability to utilize code-reuse, by allowing developers to create customized modules that can easily be integrated into the GM. By providing a common game engine logic the GM provides a core architecture that decouples applications from the Delta3D game engine. The GM provides core architecture of Delta3D, effectively binding the different elements in the game application together around a central part of the game application. It manages all actors, both static and dynamic, in the game, provides messaging and communication between objects, and finally via a rules component controls the behavior of components added to the GM.

The Delta3D framework as a whole provides a large set of classes, both

abstract and concrete, which includes functionality in a wide range of different areas of game and simulation development. The GM framework should not be seen as a separate entity in this context, but is an integrated part of the overall Delta3D framework. Instead the GM framework is comprised of a subset of the Delta3D framework, which has been designed to fulfill the specific role of providing applications with the features and possibilities mentioned previously. It is important to notice that the GM framework is the overall architecture of the solution presented in this section, and that the GM framework in addition to several supporting classes, also contains a concrete class called `GameManager`.

4.3.2 GM Framework Organization

Where the Delta3D framework is a complete application category framework, the GM component provides an application developer with a much smaller and more specific domain category style framework. The GM framework does not provide a developer with the ability to create a complete game or simulation application, but rather focuses on the specific domain of controlling game engine logic. In this project the GM part of the Delta3D framework is referred to as "the game manager framework", but in reality it might be more accurate to classify the GM as fulfilling a role, according to [Van Gurp and Bosch, 2000], rather than a framework. Here the GM represents a set of functionality, which can be used in applications, which is a subset of the overall functionality of the Delta3D framework. Since the architecture of the GM framework is not presented to the developer through a single whitebox interface, but rather through a single blackbox implementation, the `GameManager` class, the definition would not exactly follow that of a role. Instead the `GameManager` class, and the associated classes that are used with it, are in this project considered to be an independent framework, that are included within the Delta3D framework.

The GM framework is a composite framework, both providing blackbox functionality, and whitebox flexibility. While the framework provides concrete implementation of all classes, and such are ready use after only a minimal instantiation, the real power of the framework lies in the ability to inherit from the provided classes, and the override the build functionality with custom code. Figure 4.3 shows a number of selected classes from the framework. The cen-

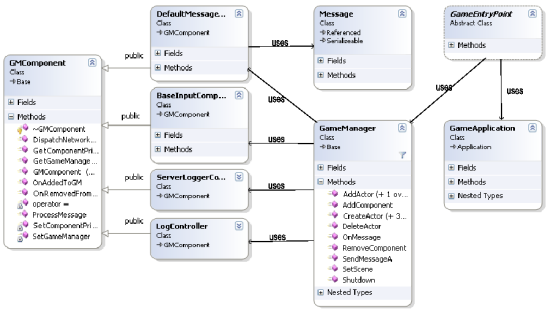


Figure 4.3: Game Manager Framework - Selected classes

tral class is the concrete class `GameManager` which handles all internal logic in the GM framework. This class publishes a number of methods for adding and

removing custom components. A high level description of the functionality is provided in the next section. The next most important, and probably most interesting class in this project, is the `GMComponent` which is the base class for all components added to the GM. The `GMComponent` class provide some build-in default capability, but is really used through the many inheriting classes, like the `BaseInputComponent` and `DefaultMessageProcessor`. Again, like the `GMComponent` these classes provide a default implementation, but is also the place to add any application specific increments to the framework. This is done by inheriting these classes, and overriding functionality as needed. Section 4.3.4 provides some details on how instantiation of the framework is done, including overriding functionality of the build-in classes.

4.3.3 GM Framework Functionality

This section presents the main parts of the GM framework in a broad perspective. The goal of investigating these elements, is to understand the internal logic of the GM framework, which will then be utilized both for creating an application for testing purposes in this project, and also primarily to understand expand and increment the GM framework with additional functionality.

The GM framework is an abstract layer which provides several components to be used in Delta3D game development. The four main components consists of:

- The Game Manager - Overall control structure.
- Messaging Passing[2] - Handles all communication.
- Game Actors - Creates dynamic player controlled contend.
- Game Components - Controls game logic.

These components are all bound together in the `GameManager` class. Figure 4.4 illustrates the relationship between the elements in the GM by showing its main components and examples of their use. For example it is illustrated that the GM holds components, which could include such components as networking or message processing. The GM maintains a list of all added and active components and all actors, both game and non-game actors. By registering with the GM, the GM tracks which actors and components that wish to receive messages from other modules in the GM, and passes these messages between modules. The GM tracks and handles low-level events received from the Delta3D game engine, passing events to interested parties, for example pre-frame and frame events.

Message Passing is provided by the GM to handle inter-object communication between interested parties in the game application. The GM exposes methods for sending and processing messages, with the intended goal of creating a central message passing interface for all actors, components as well as other GM's in a networked application. The concept behind the message passing architecture is shown in figure 4.5 where the GM maintains a central message queue, exposes a method for other modules to send messages, and in turn invokes appropriate methods on added components.

[2]The Message Passing is not really a component, but more an internal feature

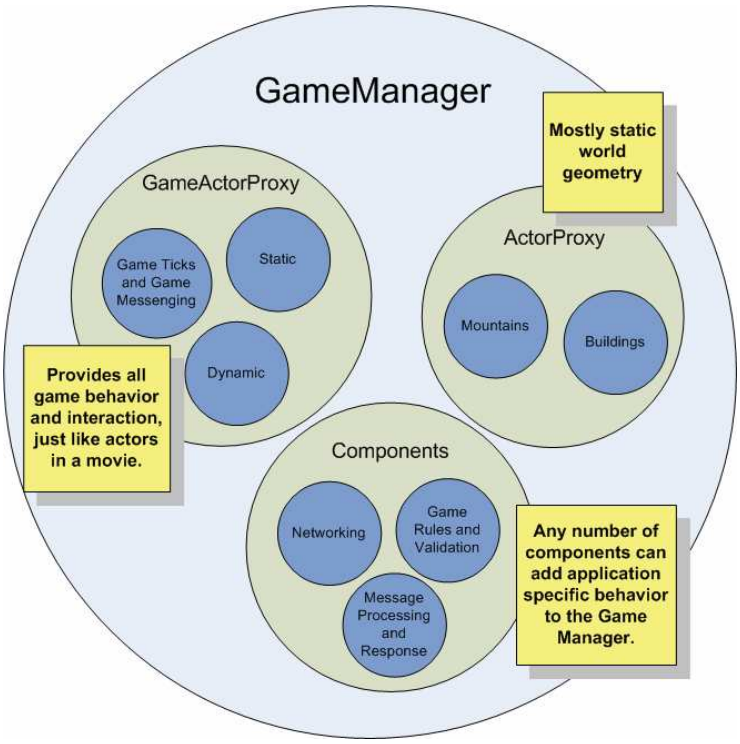


Figure 4.4: Game Manager set diagram

Messaging Architecture

A message in Delta3D is an event fired internally in the application, along with a set of data that defines the event. All messages must be registered with the GM, and must be created using a message factory. The use of a message factory ensures that all messages are registered with the GM, and also provides for an easy and efficient way of creating messages. Figure 4.6 shows a class diagram detailing the design of the message class. The MessageFactory class provides an interface for building messages, and also maintains a list of registered message types, which it uses to enforce type safety for messages. The Message class provides the actual message code. Some of the features of the message class includes:'

- Sender identification and receiver identification.
- Time stamping.
- Serializing interfaces - this allows messages to be send both over network and stored in files.
- causality information - this provides message history for determining which message caused this message to be send.

The MessageTypeEnum class holds information that can be used to identify types and classes of messages, and are used by both the message factory and by

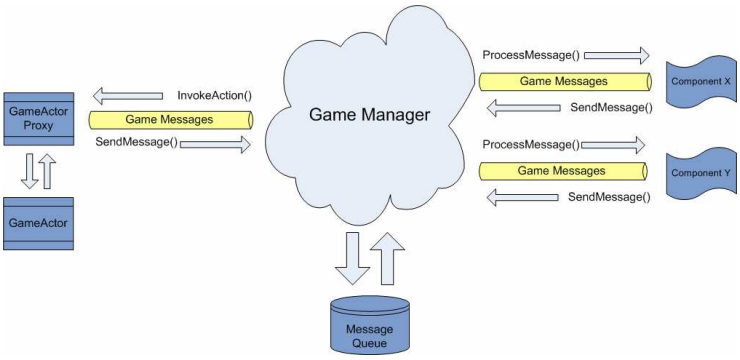


Figure 4.5: Game Manager message passing

the rules component to create and control messages. Important message classes include:

- Tick - tick events control the flow of the game by signalling updates in a regular and timely way. Each tick received by the GM signals a new update of the game should take place[3], which usually includes frame and pre-frame calculations. The concept of game updates in a game loop was discussed in [Horn and Grønbæk, 2008]. Two type of ticks are handled, a `TICK_LOCAL` and a `TICK_REMOTE`, signalling either a tick message from the local game engine, or a signal received from a networked game engine. Each tick message contains several fields with data, where the `DeltaSimTime` field in particular is used when updating actors, since it indicates time since last tick.

- Timers - Delta3D supports a number of different internal timers, which all generate events passed on as messages to the GM upon firing. Different timer message types invokes different timers and associated responses.

- A special message class is specified for message rejection purposes. This allows the GM to ignore messages received from clients or components which was not suppose to send messages, or other components to ignore messages that cannot or should not be evaluated. Upon receiving such a message, a `MESSAGE_REJECTED` message is returned to sender, which can then react appropriately to this information.

Apart from the mentioned message types, a large number of message classes are provided in Delta3D by default, primarily messages concerned with game engine logic.

Game Actor Design

The GM introduces the concept of game actors, which is a distinction from the regular actors implemented in Delta3D. Regular actors are part of the core Delta3D and can still be invoked as necessary, while game actors are a special class of actors defined to be used in conjunction with the GM. The purpose of

[3] potentially at least, it is up to the game developer to handle ticks

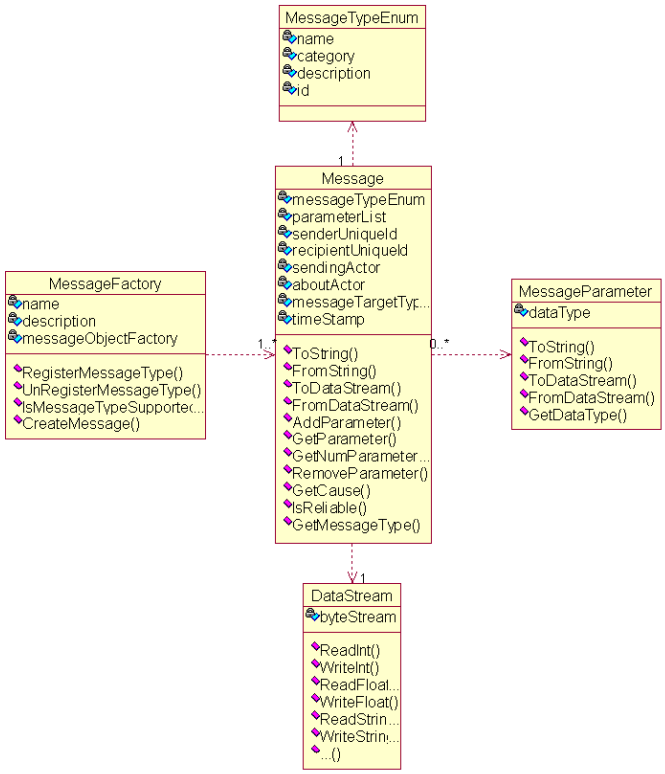


Figure 4.6: Message Class Design

introducing the game actor class is consistent with the purpose of the GM it self, to decrease complexity and increase code reuse. All game actors implements certain properties that the GM can access and control, and in addition implements a game actor proxy. The game actor proxy ensures a consistent and efficient way of invoking methods on game actors, using messages from the GM. Figure 4.7 shows the class diagram for the game actor and game actor proxy classes. The GameActorProxy class provides an interface for the GM to interact with a game actor. All game actors must implement a game actor proxy, and Delta3D recommends letting a game actor and its proxy share the same .h and .cpp files, to stress the relationship between these two classes. A game actor proxy implements a number of so-called invokables, which are a specially defined way for the GM to communicate with game actor proxies, and thereby also game actors. Each invokable is registered with the GM and take a message as argument. The invokable extract parameters from the message, and then calls methods on the game actor with the parameters. When invokables are registered with the GM, the game actor proxy specifies which message types should be passed on to the proxy and the actor.

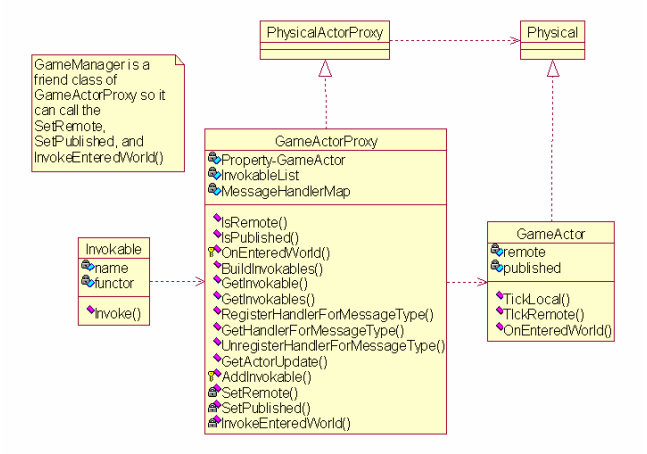


Figure 4.7: Game Actor class diagram

Component Design

An important part of the GM is the ability to design and add components, which are a sort of high level processors. Components handle activity that are to complex or not appropriate for actors to handle. The GM architecture includes a number of basic components, and allows for the design and inclusion of custom components as well. One of the central components in the GM architecture is the `DefaultMessageProcessor` (DMP), which handles the build-in message types of the GM. The DMP for example handles adding and removing actors from the GM, and handles object life cycle both for remote and local objects. Most methods in the DMP can be overwritten, which is the basis for creating customizations of games using the GM. Messages that are caught by the DMP are analyzed for types and content, and based on that information handled appropriately. Typically when a message has been handled, it is removed from the GM, but the DMP can choose to return the message to the GM for further processing. Another central component included in the GM is the `InputComponent` (IC), which is handles all types of inputs from the user interface. Generally the IC captures input from mouse and keyboard, but the GM includes several specialized components for more advanced input units. As with the DMP, once an IC captures an input event from an input unit it will analyze and handle the input appropriately. Usually the input event is then discarded, but IC's can be configured to pass the input event back into the GM for further processing.

4.3.4 Framework Instantiation with the GM

Framework instantiation of the GM framework is dependent on the particular use of the GM in an application. In general instantiation takes place by configuring blackbox elements, and glueing these elements to the application developed. Whitebox elements must be developed accordingly to their needed requirements, and then added to the application with glue code. While there is no particular method for instantiating the GM framework mentioned in the

Delta3D documentation, the obvious way of doing so, is through the use of the `GameEntryPoint` class, which is also shown in figure 4.3. The `GameEntryPoint` is an abstract class which must be implemented to configure the use of the GM. When implemented the class can be invoked with the special Delta3D class `GameStart`, which is constructed so developers do not need to implement a main method in their code. The code in the implementation of the `GameEntryPoint` class can be considered to be the glue code binding the application developed with the framework provided by the GM. A developer using the GM framework can implement whitebox elements, like the components accepted by the GM class, and can then, in the glue code, configure the blackbox GM class by adding these components to the GM.

4.4 GM Virtual Environment Test Support Tool

This section describes the design and implementation of a input capturer and playback system implemented for Delta3D. The system is implemented so as to fulfill the criteria for a test tools as described in section 3.2, which can be summarized to the following requirements:

1. Generate test input
2. Generate test output
3. Execute test cases

This is similar to step two through four in table 3.1. Apart from the three requirements, which are elaborated in the following, the system will be developed using the Delta3D Game Manager architecture. This should allow the system to be added to existing Delta3D application using the GM, and also ensure that system developed is of a reusable character, event though re-usability is of secondary concern in this project.

Requirement one is interpreted to mean that the system must be able to generate the needed input that takes the application being tested from its starting state, and through a series of states to its final state. In the implementation this requirement means that a human tester will be able to interact with the system, and that the interaction thus generated will be stored in an appropriate format, that can later be reused. This data is called the test input.

Requirement two specifies that the system must be able to generate test output. The test output is the data that is described in the previous sections as oracle information. The system must be able to store some selected data, that identifies the current application state with regards to the test case being created, in such a format that a oracle procedure can later use this oracle information as input for an oracle procedure. In this system the oracle information is stored as screen shots capturer in the virtual environment, which can later be examined by a oracle procedure. When screen shots are used as oracle information, it also becomes a requirement, that the test input generation, which was requirement one, also includes information on when and where screen shots are obtained. This is solved by creating the system, such that when generating test output, which is done in-application by a human test case designer, the information on when and where a screen shot was taken is stored in the test input data.

The third requirement is that system must be capable of replaying one or more test cases, which consists of a set of generated test input along with the selected test output. The system must be designed so that it will reply the test input, including the stored information on when to generate oracle information, and store the captured data as oracle information, in a way that makes it possible for the oracle procedure to comparer the oracle information with the reference data.

4.4.1 Design

The GM VETS tool is designed as an application specific increment of the GM framework, but also designed such that the implementation is so general that it could be included in the framework, as an internal increment, for future releases. A design choice was made that for this project it was sufficient to capturer, log and replay keyboard from the user, and that input from other devices, primarily the mouse, could safely be ignored. The GM framework also has build functionality for capturing mouse input, and if needed, an additional component for logging mouse input could be created. While it probably would be possible to capturer keyboard and mouse input at an even lower level than the Delta3D framework, for example by implementing the functionality in C and running the program externally from Delta3D, it was decided that the functionality would be implemented as part of the GM framework. This is done to ensure that the logger fits into the framework in the case of future updates, and also to allow for more nuanced logging. By including the logging functionality into the framework, it is possible for future users to easily override functionality and adapt the logger for their specific purposes. Based on the analysis of the GM framework

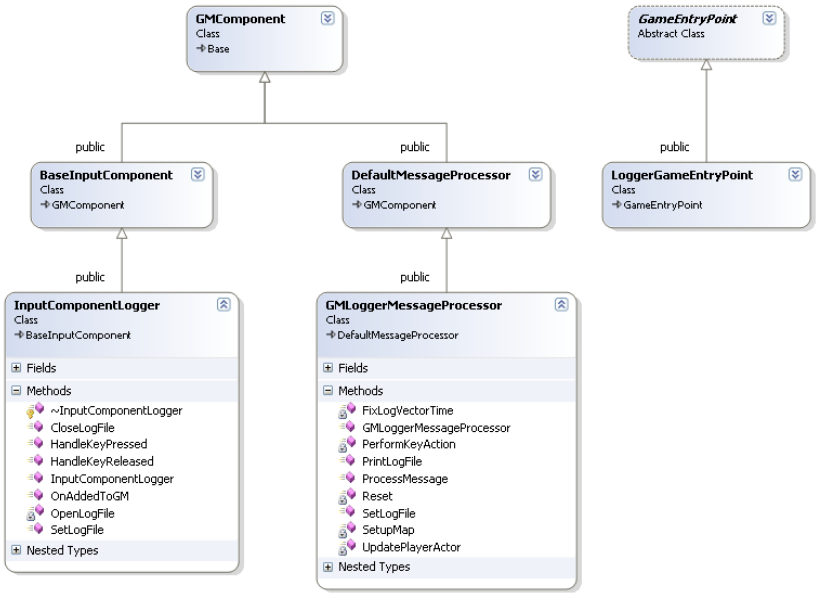


Figure 4.8: VETS class diagram

in section 4.3 it is clear that to design and implement a working input logging

functionality, which is compatible with the current and future Delta3D versions, creating components for the GM would be an ideal choice. By combining the flexible whitebox features of the `GMcomponent` class with the easily accessible features of the blackbox `GameManager` class it is possible to create customized functionality, application specific increment, which can be included easily into the GM architecture. Figure 4.8 shows a class diagram of the most important class in the GM Input Logger design, except perhaps the actual `GameManager` class. To implement the needed functionality two separate components are designed, `InputComponetLogger` for logging and storing input from the keyboard, and `GMLoggerMessageProcessor` for reading stored log material and then processing and recreating the input events. Both these components are created by extending existing implementations of components inheriting from the `GMComponent` class.

The InputComponetLogger

The `InputComponetLogger` inherits from the build-in `BaseInputComponent` class, which allows the class to take advantage of the already declared virtual functions: `HandleKeyPressed` and `HandleKeyReleased` which is invoked by the `GameManager` when receiving input events. These methods are redefined in the `InputComponetLogger` implementation, to examine and, if appropriate, log the data extracted from received events of the type `KeyPressedEvent` and `KeyReleasedEvent`. In addition to the event data, which includes identification of the key used, the simulation time since startup of the key event is also logged. To handle common files system operations methods for configuring, opening and closing output streams are also added to the component. Finally, to simplify usage of the component the virtual method `OnAddedToGM`, inherited from `GMComponent` class, is implemented. The `OnAddedToGM` is called when a component is added to the GM, and a suitable implementation is found in the component. The `InputComponetLogger` class fulfills item one of the requirement list by allowing the application to generate and capturer test input.

GMLoggerMessageProcessor

Like the `InputComponetLogger`, the `GMLoggerMessageProcessor` is based on an existing implementation, inheriting from the GM component `DefaultMessageProcessor`. The virtual `ProcessMessage` method is implemented to ensure that all messages are captured and processed by the class, so that any relevant messages received can be handled. In the particular implementation for the GM Input Logger used in this project, the messages of interest is limited to a single class of messages, timer elapsed events, so rather than implementing a full message processor, it would also have been possible to implement a simple component, and registering that component as a receiver for timer elapsed messages. As a design decision the message processor type component was selected, since this would make it more obvious that the purpose of the component was to handle messages, and future implementations could more easily handle other types of messages if necessary. Apart from the process message method a number of additional methods were included in the component for easier handling of log files and data. The `GMLoggerMessageProcessor` implements functionality to fulfill the requirement indicated as number three in the requirement list.

In addition to the actual replay functionality, the `GMLoggerMessageProcessor` class is also responsible for implementing the second requirement, test output generation. By design a virtual function is declared, where a simple image capturer functionality is implemented. The `ProcessMessage` function is implemented so it will call this function when receiving a signal that output must be generated. This allows for both blackbox usage, where the default implementation is used, and for more flexible whitebox use, where a developer can override the output capture function as needed.

4.4.2 Testing with a Delta3D Application

To test the GM Input Logger a sample Delta3D application using the GM framework was implemented, and incremented with Input Logger classes. Figure 4.9 shows s screen shot from the GMTank application which uses the GM framework architecture. Here a user navigates the tank around in the virtual environment

Figure 4.9: The HoverTank sample application

using the keyboard, and likewise controls camera position and rotation. The tank can move around like it would be expected for a vehicle: forward, backward and turning right and left. In addition the application features a engine on/off function, where the tank is incapable of movement will the engine is off. When the engine is on movement becomes possible, and this indicated in the virtual environment by letting an animation of dust, simulating engine exhaust, appearing at the tanks position.

Design

Figure 4.10 shows the complete class diagram for the GMTank application, including application specific classes, and the class created for the VETS tool. The central class in this application is the `TankGameEntryPoint` class which is

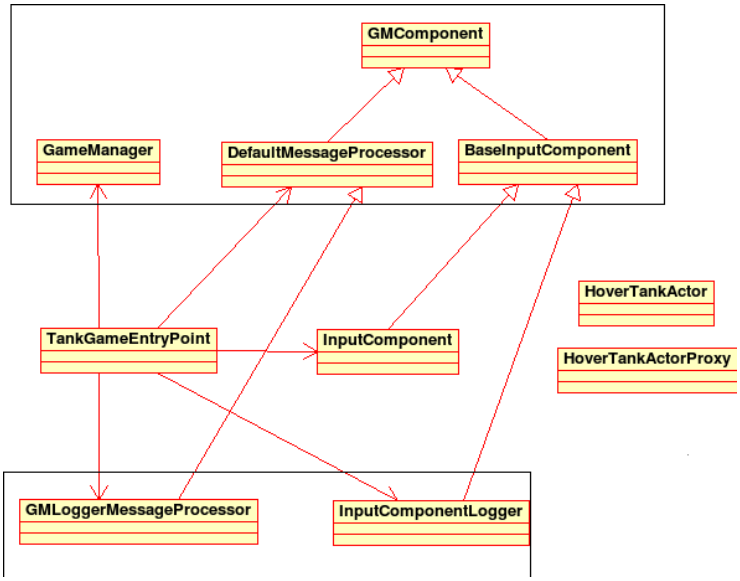


Figure 4.10: The GMTank sample application class diagram

responsible for instantiating the GM framework used in the application. The `TankGameEntryPoint` class uses both the `GameManager` class and the `DefaultMessageProcessor` directly as a blackbox components, but also uses the two whitebox components, implemented as part of the framework increment done in this project, `GMLoggerMessageProcessor` and `InputComponetLogger`. The `InputComponent` is an application specific increment done for controlling input in the application. Finally th class diagram shows the actor class `HoverTankActor` and its associated `HoverClassActorProxy`, which are not directly connected to the other application classes. This separation is part of the original GM framework functionality, where rather than binding the actors to the application, the actors are bound in a map[4], and can then be inquired through the map functions in the GM.

Tests

This section presents and evaluates a number of tests performed to validate the functionality of the developed VETS tool. The tests have been done on the GMTank application by first recording input for a test case, and selecting a number of scenes for test output. The build-in output generator, in the form of a simple screen shot capturer is used, and screen shots are saved for reference material. The GMTank application is then executed with the test case as input and the oracle information generated is compared to the reference screen captures.

First, to illustrate the functionality of the VETS tools input capturer function, listing 4.11(a) shows a partial sequence of the captured input to the GMTank application. This listing shows the time, key and action (press or release) stored in a simple text file. The input is provided by a tester executing the

[4]A map is a Delta3D "level" and can be designed in a separate stand-alone tool included with Delta3D.

```
1  3.78803:65361:1
2  4.95433:65361:0
3  5.05428:65361:1
4  5.53745:65361:0
5  5.92064:65363:1
6  6.25388:65363:0
7  6.43712:65362:1
8  6.57032:65362:0
9  6.72024:65361:1
10 6.88685:65361:0
```

(a) Test

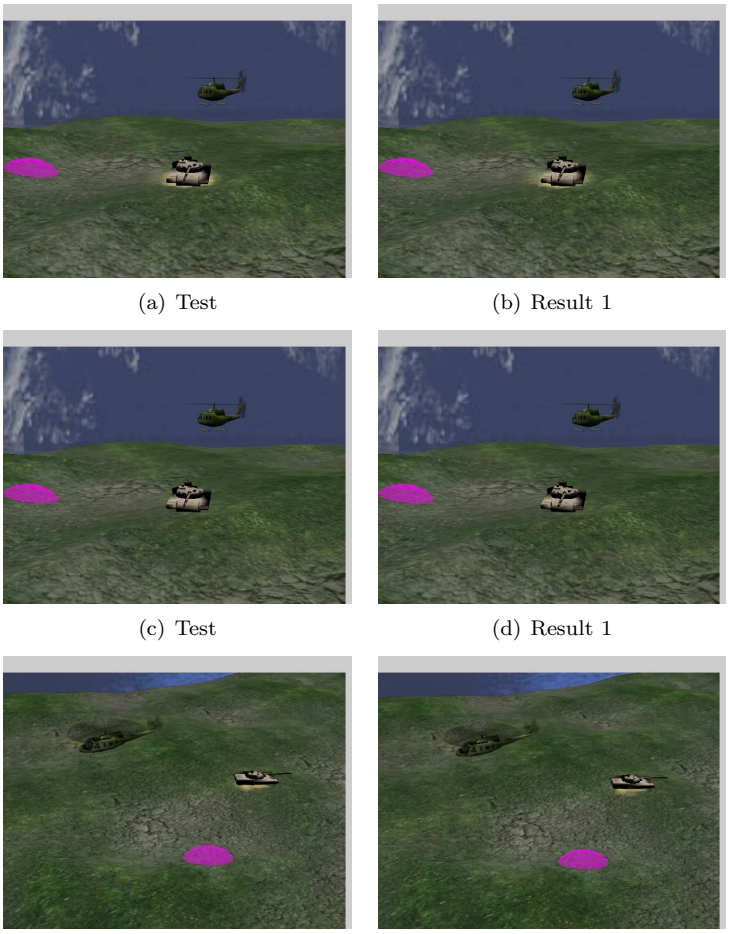
```
3.79544:65361:1
4.96189:65361:0
5.06168:65361:1
5.54488:65361:0
5.92807:65363:1
6.26125:65363:0
6.44452:65362:1
6.57769:65362:0
6.72766:65361:1
6.89424:65361:0
```

(b) Result 1

Figure 4.11: VETS input file and associated output log

application and interacting with the application through the keyboard. The sequence in listing 4.11(b) shows the same input sequence, but this time captured as a result of executing the application with the data in listing 4.11(a) as input. Figure 4.12 shows the reference images captured during the initial manual execution (left column) side by side with the oracle information (right column) captured by running the application with the input from the manual input. The figures serves to confirm the tester that the moving actor, the tank, which is controlled by the keyboard has moved as expected. In addition the camera transform, which is also controlled by the keyboard, can be visually confirmed to be correct.

While the images in figure 4.12 can be used to visually confirm the correctness of the application, in reality the images are not completely identical in terms of a pixel by pixel comparison. This is the result of several factors, both Delta3D rendering methods, so-called z-buffer fighting, and not least of all, inaccuracies in the replay timing functionality. The replay functionality is accurate enough that the level of details that can be seen by the human eye in most cases will consider the captured images identical. Even when not completely identical, the human oracle procedure can easily estimate whether the captured oracle information validates the correctness of the application or not. The inaccuracy can be spotted in figure 4.12(e) and 4.12(f), where close inspection reveals the images not to be completely identical. An example of a situation, where there is a more obvious inaccuracy captured is shown in figure 4.13, which is part of a test case performed by the test tool. Here it is clear that the two images are not exactly identical. Depending on what the human tester was expecting, it is his or her responsibility to evaluate if this test confirms correctness or not. If the accuracy of the tests are an issue for the testing procedure, for example if the Image Analyzer is employed for image comparison, a simple solution can be used to work around the problem. When the test case has been capturer during the input generation and a first execution of the application has been done, the output of this test can be used as reference screen shots for further testing. The inaccuracies of the replay system can be negated this way, since this is a constant factor in all executions. Figure 4.14 shows the output from the GMTank application executed with the same test input as before, the test input also shown in part in listing 4.11(a). In the right column the output of the test execution is shown, just as in figure 4.12, but rather than using the original



(a) Test (b) Result 1

(c) Test (d) Result 1

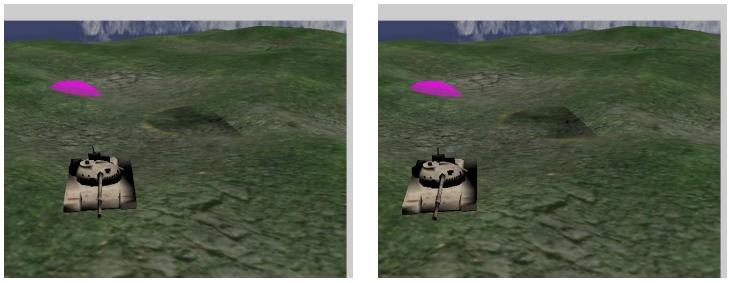
(e) Test (f) Result 1

Figure 4.12: Reference screen shots side by side with oracle information

reference screen shots in the left column, the references has been exchanged with the output from the first test. If a human oracle procedure has confirmed the correctness of the application based on the images in the left column, these images can be used as correctness measure in the following tests.

4.4.3 Conclusion

The VETS tool was designed with the intention of implementing a functional tool which could assist a Delta3D tester in performing the virtual environment procedure test steps found in table 3.3. The result of the implementation has been tested with a Delta3D application, as shown in the previous section, and found to be a usable tool, which can help automate steps in the testing procedure. While it is impossible to guaranty that the tool can be used for every possible implementation of a Delta3D application, it has been purposely designed so that it could be implemented as an internal increment to the GM framework in Delta3D. By honoring the design and ideas behind the GM Framework in



(a) Reference (b) Test

Figure 4.13: Inaccuracy in application execution

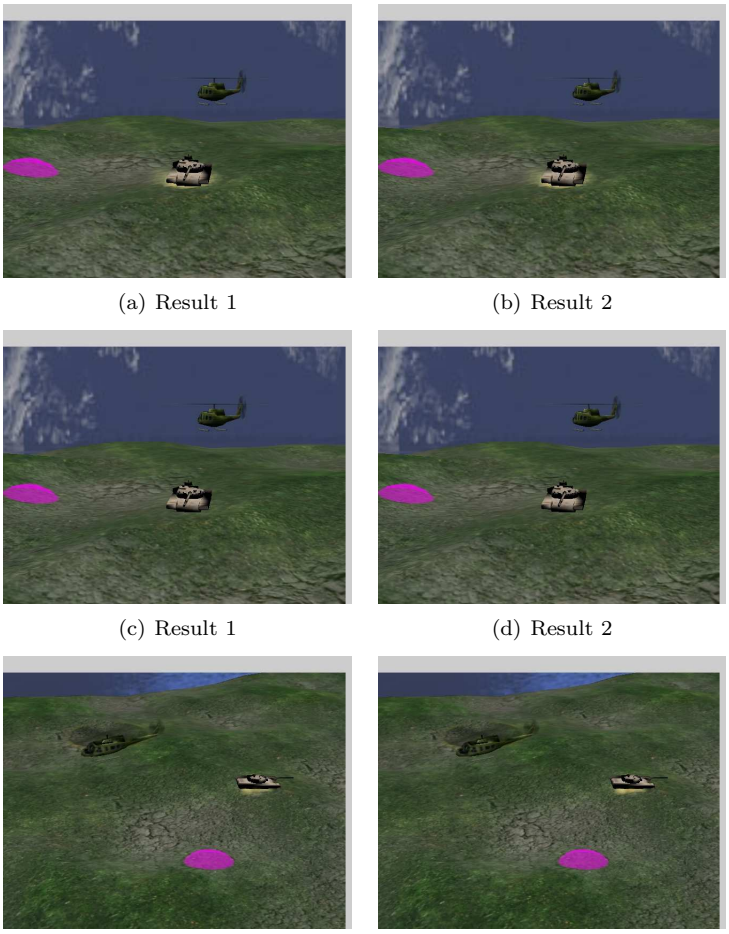
the design of the VETS tool, the tool can easily be integrated into new and existing applications that are using the GM Framework. The tool itself allows for both easy blackbox use, with a few simple configurations, and for more flexible whitebox use where developers implement custom functionality as needed. This is both in line with the recommendations by [Van Gurp and Bosch, 2000], and also very similar to the existing functionality found in the GM Framework. The VETS tool is not perfect, as it was illustrated with the perceivable image inaccuracies shown in the previous sections, which depending on the test scenario can be of minor or major concern. For this project a work-around was employed, but a more advanced and precise implementation of the tool would definitely be a major part of a future work on the tool.

4.5 Delta3D AAR Architecture

The Delta3D framework has a build-in logging capability, called the After Action Review (AAR) system, which is implemented as part of the GM framework role. The AAR system allows output and storage of game messages to a log file, debug console or similar. If the log messages are stored in a log file of specific (binary) format, the AAR system allows replay of the captured messages at a later time. This record and playback capability is implemented as an integrated part of the AAR system in the Delta3D framework. AAR is typically used as a verification and validation system, for example by allowing instructors or trainers to review a particular simulation after its completion. In the following sections the AAR system functionality is described in more detail. Next, the potential use of the AAR system in relation to the previously discussed VETS system is evaluated.

4.5.1 AAR Components

Figure 4.15 shows an overview of the AAR logger components used with the GM framework. In the figure the Network block is optional, since the AAR system also works on in a local application. In the diagram, the Game Manager component, serves as the central entry point for all AAR related actions. The AAR system is broken down into two separate components, both of which are loaded and controlled by the `GameManager`. The `ServerLoggerComponent` class performs the actual logging to, and reading from, a stream, while the `LogController` component is used to control the operation of the AAR system.



(a) Result 1 (b) Result 2

(c) Result 1 (d) Result 2

(e) Result 1 (f) Result 2

Figure 4.14: First set of oracle information side by side with second set of oracle information

The ServerLoggerComponent

In figure 4.15 the arrows from the Server Game Manager to the Server Logger component indicates that all messages are sent from the Game Manager to the logger, which also sends back updates to the Game Manager if necessary. The Server Logger component is the primary component of the AAR system, and performs all of the work necessary to implement the AAR system. It communicates with the log stream or log file as needed, so no direct interaction with these components are done directly from the application. Like other components in the GM framework the Server Logger component is completely driven by messages, rather than actual method calls through an API. The primary class of the Server Logger component is the `ServerLoggerComponent` class shown in figure 4.16. This class receives all game messages sent to the `GameManager` it is associated with. It records all game messages as well as processes game messages that are specifically sent to the `ServerLoggerComponent` to control it.

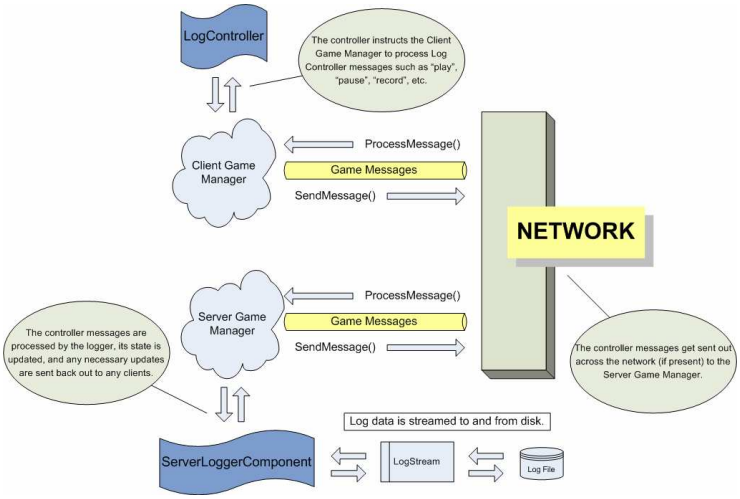


Figure 4.15: AAR Logger Components

These messages allow the application to control all aspects of the Server Logger component, for example whether the AAR system is recording the current simulation or alternatively doing a playback of a previous recording. Controls also include the ability to change log file, insert tags and mark key-frames. Each application should only have one Server Logger component associated, so in a distributed application the Server Logger component should be added to the server part of the application.

Apart from the `ServerLoggerComponent` class the diagram in figure 4.16 shows four other classes used in the Server Logger component. The `LogTag` class allows an application to add information to a recording of a simulation, by inserting a tag with a title, a description and a time stamp which can then be used to illustrate particular areas of interest in the recording. The `LogKeyFrame` class is used to store information about a particular point in time. It stores the map used in the application, including information about all actors in the map, as well as a time stamp which allows the application to shift between specific key frames in a recoding. The `LogStatus` class is used for book-keeping of various aspects of the Server Logger component, and is primarily responsible for keeping the Log Controller and the Server Logger synchronized. The `LogStream` class is a virtual interface to one or more log files, allowing streaming of the recording to such files. The GM framework comes with one particular implementation of the `LogStream` interface which uses a specified binary format for storing recordings by the AAR system.

The Log Controller Component

The Log Controller component is controlled by the application using the AAR system, and sends messages to the Game Manager indicating if the AAR system should record or playback messages, or perform other AAR functions. The Log Controller keeps automatic synchronization with the Server Logger using LogStatus class. The Log Controller component class diagram is shown in figure 4.17. The single LogController class in the diagram provides and API for using

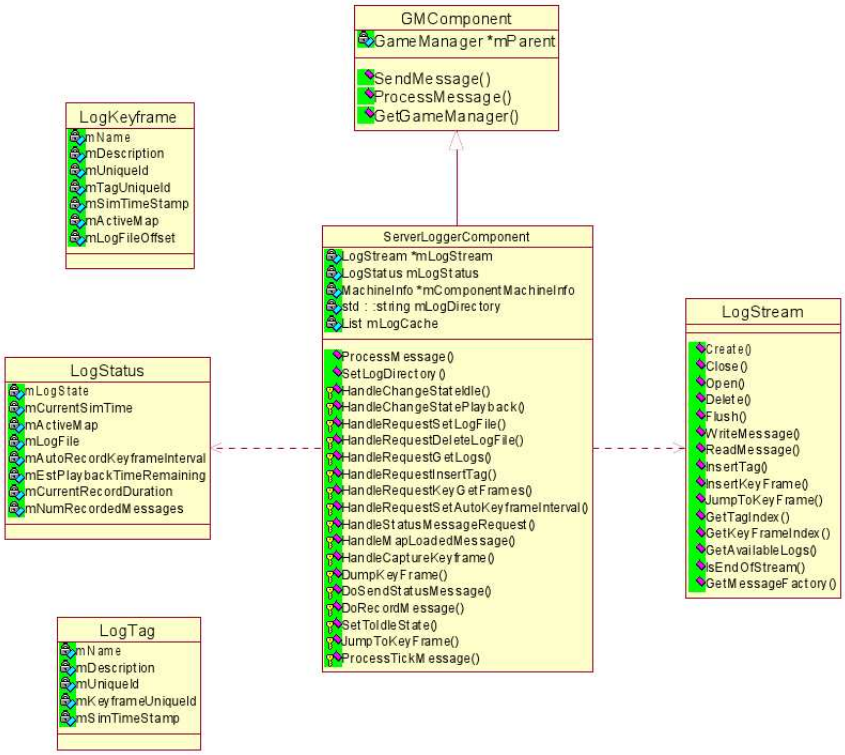


Figure 4.16: ServerLoggerComponent Class Diagram

the AAR system from an application. Most of the methods provided by the LogController class is wrappers that creates and sends appropriate game messages to the Game Manager and on to the associated Server Logger component.

4.5.2 Using AAR as a Test Support Tool

While the final implementation of the project automation tools for the testing procedure, do not include the use of the AAR system, the system was initially considered for this project. An application using the same map as the GMTank application was developed, and the AAR system integrated in this application. Figure 4.18 shows the application with the AAR system embedded. Like the GMTank application, this application was developed using the GM Framework, which allows for the use of the various AAR components, as presented in the previous section. The next section presents consideration on advantages of using AAR, while section 4.5.2 presents the disadvantages found, and ultimately the reason why the AAR system was not use in the VETS tool.

Advantages of AAR

With the AAR system in place, it was indeed possible both record and replay execution of the application. In the developed application the different function-

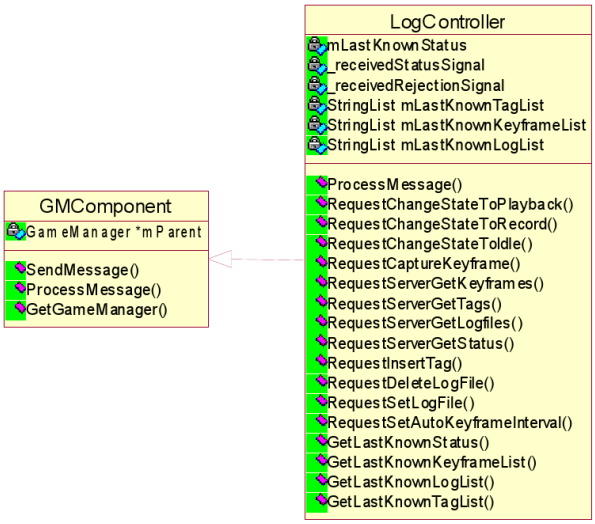


Figure 4.17: LogController Class Diagram

alities of the AAR system was bound to keyboard keys, so the user executing the application could start, stop and pause both recording and replay directly from the application, rather than running the recorded continuously, as the VETS tool do. Figure 4.18 shows a recording in *pause* mode. When doing replays it

Figure 4.18: AAR allows pausing and stopping replays

was possible to use several very practical features exposed by the AAR to control the replay, most noticeably the ability to jump between tags in a recording, which would allow a test to skip over non-important parts of a recording. An example of using keyframes, which indicates "jump-points" in a recording is shown in figure 4.19, where the number of keyframes in the recording is shown, together with the time of the next keyframe. Parts that were replayed could have the internal time scale in Delta3D increased and decreased, which can also

Figure 4.19: AAR allows skipping to keyframes

be utilized when a human tester is need to watch recordings for correctness testing. Figure 4.20 shown an example with the internal time scale increased to

Figure 4.20: AAR allows changing the internal time scale

allow faster playback. These three features would make the AAR system very interesting in a testing system such as the one developed in this project.

In addition to the general usefulness of the above features, the tagging feature described in the previous section, could potentially be used to create a system much like the design described in section 2.3.4. The tagging feature could be used to allow a tester to create indicates when the application reaches a state that must be checked. By creating a test runner application, as the one shown in figure 2.5, it would be possible to rerun recorded executions, and create a `MessageProcessor` component for the `GameManager`, that listens for messages send when the execution reaches a tag, and in response performs testing.

Disadvantages of AAR

In spite of the number of advantageous features in the AAR system, several disadvantages where discovered during the implementation and testing of the system, which ultimate led to the system not being utilized in the VETS tool.

The first drawback of using the AAR system is that it might require reconstruction of an application. While the system used in the VETS tool records input events directly, the AAR system records event in a different fashion. The system must be associated with a message processor, that is added to the `GameManager`, and records events and messages that are received by this message processor. This has the consequence that the AAR component only records messages which are passed on to the message processor that has been associated with the AAR component. The `GameManager` sends event to all registered handlers, in the order of their priority, and since some components does not send messages that they have received and handled, back up to the `GameManager`, this might result in some events not getting through to components with lower priorities. This could be solved by giving the message processor associated with the ARR system top priority, but since this in general discouraged, it might not be a viable solution. Figure 4.21 shown an example of a message path from

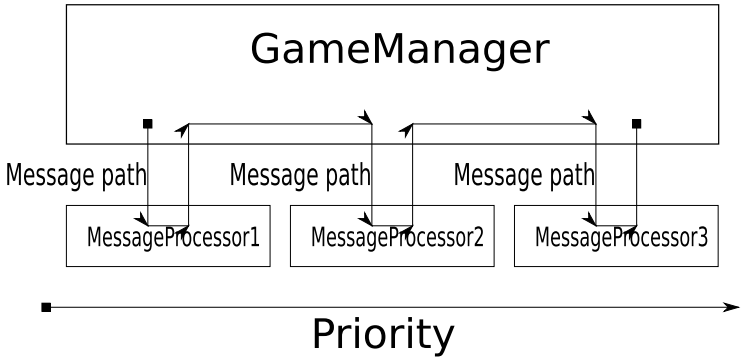


Figure 4.21: GameManager message path

the `GameManager` through several `MessageProcessors`. Another solution to this particular problem is to make sure all message processors are "fall-through" with regards to messages. This requires the tester to have the ability to change in the construction of the other components, and it might also present new problems when all messages are passed to receivers they were not originally intended to reach.

Another drawback discovered with AAR system as a test support tool, is the possibility of changes occurring in other component, which would be a natural consequences of development, which is likely to break old recordings. Since the AAR system records events and messages and stores these for playback later, changing the way an actor handles a message may render all previous recordings invalid. While this is also the case of the VETS system developed, it was thought in this project, that a change of message handling functionality in an actor, were much more likely to take place, than changing the input keys of an application.

Finally, and more insubstantial disadvantage with the AAR system, is the general lack of high quality developer documentation of the system. The im-

plementation used in this project for evaluating the AAR system, is based on a sample application demonstrating AAR. While this application works fine, it is very hard to reconstruct the functionality in another application, since much of the code is not documented, or self describing. Together with a general shortage of documentation, it is hard to do a real full scale test and evaluation of the complete AAR system.

4.5.3 Conclusion

The conclusion after testing the AAR system in this project is, that the system could be very interesting in a test automation or test support tool, as the one developed in this project. The AAR system has some obvious, and some not so obvious, problems that would have to be explored in more detail, requiring intimate knowledge of the AAR system, but if those could be overcome, the component based system could become very practical. The system employs some advanced features that would be interesting to have in a test support tool, and is already tightly integrated in Delta3D, which is an advantage in terms of stability and in terms of interoperability with the Delta3D framework in general.

The exploration and testing needed to develop further on a solution employing the AAR system was abandoned in this project, in favor of focusing on a simpler, less feature enhanced, tool, which could be deployed and tested immediately. Future work could either try looking the AAR system in more detail, or could try to implement AAR features in the VETS tool.

Chapter 5

Testing 3D Graphics and Simulation Engines

This chapter presents a central framework extension, the image collector, which we have implemented to enable regression testing in Delta3D. First, an overview of 3D graphics and simulation engines used in todays virtual environments is presented. Hereafter, a general introduction to the concepts found in simulation engines, or as they are more commonly referred to, game engines. This is followed by a section introducing an actual game engine, the open source Delta3D engine, which has been used to implement a significant part of the software created in this project. Finally, an introduction to scene graph techniques in general along with the Open Scene Graph implementation used in Delta3D is presented.

5.1 Game Engines

This section gives an introduction to game, or simulation, engines in general and introduces some of the important subsystems and responsibilities in a game engine. An important part of this thesis is about 3D graphics in virtual environment applications and how the graphics contained in such applications is tested and verified to be correct. To accomplish this task it is necessary to investigate on of the core software components that constitutes a 3D simulator, namely the graphics engine. There exists graphics engines for which the sole purpose is to create and render 3D graphics models created in modeling applications like Autodesk 3ds and Blender, just to name a few. Although, such engines exists they are rarely used in isolation. Instead more compiled engines are used. Such engines are known as game engines.

Game engines, or more generally a real-time computer graphics engine, is a complex entity that consists of more than simply a rendering layer that draws triangles. A game engine must deal with issues of scene graph management as a front end that efficiently provides the input back to the renderer. The engine must also provide the ability to process complex and moving objects in a physically realistic way. The engine must support collision detection, curved surfaces as well as polygonal models, animation of characters, geometric level of detail, terrain management, and spatial modeling.

A game engine is responsible for managing the data and artistic content of the game and deciding what to draw on the screen and how to draw it. The decisions are made at both a high and a low level. The high level decisions are handled by the game AI and by the scene graph management system. The low level decisions on what and how to draw is determined by the renderer. The responsibility of the renderer can according to [Eberly, 2001] be summarized to the following:

1. Camera model
2. Culling and Clipping
3. Rasterization
4. Animation
5. Level of detail
6. Terrain

5.1.1 Camera Model

The camera model ensures that processing of objects only occurs if the objects is in a region of space called the view volume. All object that are completely outside the view volume are not processed, such object are said to be culled. All object totally inside the view volume are processed for display on the screen. Objects that intersects the boundary of the view volume must be clipped against the boundary, then processed on the screen. The display process includes projection onto a view plane. Moreover, only a portion of the view plane can be displayed on the screen at one time. A rectangular region of interest, called a viewport, is selected for displaying. Most game engines uses a so called perspective projection, where an infinite pyramid is formed by the eye point as a vertex and four planar sides, each side containing the eye point and an edge of the viewport. If the pyramid is limited by two planes, both parallel to the view plane, the resulting view volume is called a view frustum. The parallel plane closest to the eye point is called is the near plane and the plane farthest from the eye point is called the far plane. The combination of an eye point, a view plane, a viewport, and view frustum is what constitutes a camera model, which is shown in figure 5.1.

5.1.2 Culling and Clipping

Culling and clipping of objects reduces the amount of data sent to the rasterizer for drawing. Culling refers to eliminating portions of an object, possible the entire object, that are not visible to the eye point. For an object represented by a triangular mesh, the typical culling operations determine which triangles are outside the view frustum and which triangles are facing away from the eye point. Clipping refers to computing the intersection of an object with the view frustum, and with additional planes, so that only the visible portion of the object us sent to the rasterizer.

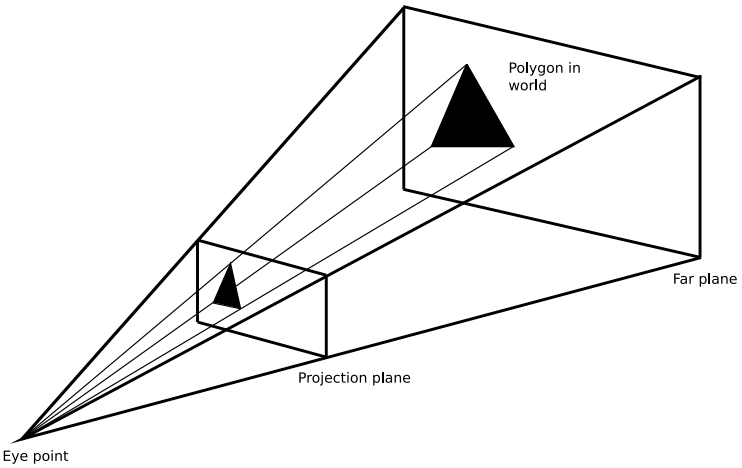


Figure 5.1: View frustum showing a polygon projection from world space to screen plane.

Culling

Object culling involves deciding whether or not an object as a whole is contained in the view frustum. If an object is not in the frustum, there is not point in consuming CPU cycles to process the object for the rasterizer. Typically, the application maintains a bounding volume for each object. The idea is to have an inexpensive test for non-intersection between bounding volume and view frustum that can lead to quick rejection of an object for further processing. If the bounding volume of an object does intersect the view frustum, then the entire object is processed further even if that object does not lie entirely inside the frustum.

Clipping

Clipping is the process by which the front facing triangles of an object in the world are intersected with the view frustum plans. A triangle either is completely inside the frustum (no clipping necessary), is completely outside the frustum (triangle is culled) or intersects at least one frustum plane. In the last case the portion of the triangle that lies on the frustum side of the clipping plane must be calculated. That portion is either a triangle itself or a quadrilateral that is partitioned into two triangles. The triangles in the intersection are the clipped against the remaining clipping planes. After all clipping planes have been processed, the renderer has a list of triangles that are completely inside the view frustum.

5.1.3 Rasterization

Rasterization is the process of taking a geometric entity in screen space and selecting those pixels to be drawn that correspond to the entity. The standard objects that most engines rasterize are line segments and triangles, but rasterization of circles and ellipses are also possible.

5.1.4 Animation

An important responsibility of game engines are to provide support for animation. The need for displaying characters, vehicles and other objects in various positions at different points in time is often found in games and simulations. These requirements are fulfilled by adding support for animation in the game engine.

Modern 3D games has scenes composes of 300000-500000 polygons and in extreme cases the scenes extend one million polygons [4]. Such detailed scenes require significant processing power for real time rendering. Smooth animation requires approximately 30 frames per second [Kenneth E. Hoff, 1997] with such complex scenes this frame rate can be hard to achieve with a single CPU responsible for both AI processing and display rendering. To overcome these restrictions one or more graphics processing units (GPUs) can instead be used to relieve the CPU from spending cycles rendering graphics and leave this to the GPUs. The article [Sibai, 2007] analyzes the scaling of the 3DMark benchmark with CPU frequency, number of CPUs, number of GPUs, and number of threads supported by the hardware. The results reveal that the benchmark scales well indicating that 3D games and 3D simulators if implemented with multiple physics and AI threads should show good scaling too on multi-CPU and multi-GPU platforms.

In character animation, where an articulated figure changes position and orientation over time. The quantities to be controlled are the local transformations at the joints of the figure. Two standard approaches to animating a character are key frame animation and inverse kinematics.

Key Frame

Key frame animation requires an artist to build a character in various poses; each pose called a key frame. Each key frame changes the local positions and local orientations of the nodes in the hierarchy. When the character is animated, the poses at the times between key frames are computed using interpolation.

Key frame animation is effectively an interpolation of translational and rotational information over time. Interpolation of translational information is simpler compared to interpolation of rotational information. A node to be used for key framing has a local translation and a local rotation, just like any other node in the system. The transformations are procedurally updated using a key frame controller. The controller is an implementation of linear interpolation or spline interpolation

One potential problem with key frame animation is that the local transformations at the nodes are interpolated in a relatively independent way. Interpolation at one node is performed independently from interpolation at another node, which can lead to artifacts, such as stretching of character components that normally are considered to be rigid. For example, the local translations of a shoulder node and elbow node are interpolated independently, but the length of the arm from shoulder to elbow should be constant. The interpolation do not guarantee that this constraint will be satisfied.

Inverse Kinematics

An alternative method for animation is to use inverse kinematics. Constraints are placed at the various node - constraints such as fixed lengths between nodes or rotations restricted to planes with restricted ranges of angles. The only interpolation that needs to occur is at those nodes with any degree of freedom. For example, an elbow node and wrist node have a fixed length between them, and both nodes have rotations restricted to planes with fixed range of angles. A hand node is attached to the wrist and has three degree of freedom. The hand can be moved to some location is space; the wrist and elbow must follow accordingly, but with the mentioned constraints.

5.1.5 Level Of Detail

Rendering of a detailed and complex model that consists of thousands of triangles looks good when the model is near the eye point. The rendering of such a model is often time consuming, but the time cost is compensated by the smoothness and visual quality of the model. However, when the same model is far from the eye point, the detail is not that noticeable because the screen space coverage of the rendered model might only be a few pixels. In such a situation, the trade-off in computational time versus visual quality is not worth the effort. A common approach to dealing with such issues is using a technique know as level of detail (LOD), which can be implemented in several ways.

Figure 5.2 shows the basic principle of LOD. Once a bounding volume's set of polygons is determined to be in view and therefore should be drawn, then another computation must be made. The purpose of this calculation is to determine how many pixels the final set of polygons actually cover on the screen. The main reason is that it is unnecessary to draw a large number of polygons if they only cover a handful of pixels. The LOD decision presupposes

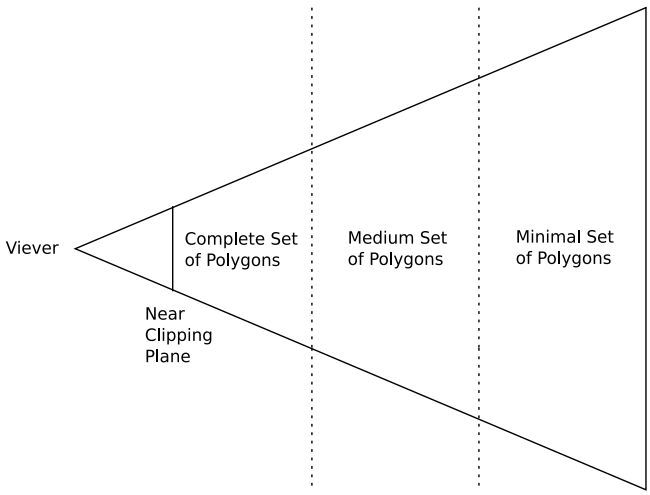


Figure 5.2: Levels of detail and the view volume.

that there are multiple sets of polygons to choose from. One choice is to draw the complete set of polygons assuming that all objects are close to the viewer,

while another choice might be to draw half the polygons reasoning that all objects are a medium distance from the viewer, and yet another choice might be to draw for instance one-fifth of the objects assuming that all objects are far from the viewer.

Sprites And Billboards

The simplest form of LOD involves two-dimensional representations of three-dimensional objects, called sprites or billboards. These are prerendered images of three-dimensional objects. The idea is that the time it takes to draw the image as a texture is much less than the time to render the object. In a three-dimensional environment, sprites are useful for software rendering simply because of the reduction in the time to draw. However, sprites can be spotted in a rendering if they represent objects that are close to the eye point or if the eye point moves. The image gives the impression that the object is not changing correctly with eye point location or orientation. The visual deviation due to closeness to the eye point is softened if sprites only are used for distant objects, for example, trees drawn in the distance.

The visual anomaly associated with a moving eye point can be rectified in two ways. The first way is to have a set of prerendered images of the object calculated from a set of predefined eye points and orientations. During application execution, a function is used to select the appropriate image to draw based on the current location of the eye point. The second way is to allow a single prerendered image to change orientation depending on eye point location and orientation. In this setting the sprite is called a billboard [Eberly, 2001] and [Shade et al., 1998].

Discrete Level Of Detail

Another approach to a LOD solution is to construct a sequence of models where the triangle count decreases over the sequence. The sequence is assigned a center point that is used as a representative of the centers of all the models. The model with the largest number of triangles is drawn when the LOD center for that model is close to the camera. As the center point moves further away from the camera, at some preselected distance the current model is replaced by the next model in the sequence. The word discrete refers to the fact that the number of models is a small finite number. The advantage of discrete LOD is the simplicity of the implementation. The disadvantage is that an artist must build all the models.

Continuous Level Of Detail

An alternative to discrete LOD is continuous level of detail. The two approaches have a lot in common, but a the advantage of using a continuous LOD algorithm over a discrete LOD algorithm, is that no additional graphical work is required to build additional models of the scenes.

LOD rendering for any mesh can be described as the process of generating a finite number of representations of the same mesh, each at a different LOD. Representations of three-dimensional objects that have high degree of detail require polygons compared to three-dimensional objects with less degree of detail,

which require less polygons. At any given time an algorithm would determine which of the pre-computed mesh representations that should be used based of the desired LOD. Switching between representations can produce a visual popping effect, since the there exists a possibility that more detail suddenly appear or disappear in a narrow time step.

Continuous LOD is a LOD approach which can determine exactly how many polygons to use for requiring a desired LOD, where the LOD is specified in a pre-selected interval. Continuous LOD algorithms eliminate the popping effect when changing the desired LOD since the changes in polygon usage are gradual. Real-time continuous level of detail is any algorithm that renders a mesh while allowing a user to dynamically modify the desired LOD for each frame at run time. For example, a real-time continuous LOD algorithm for height fields is one that allows a character controlled by the user to navigate around a terrain while continually rendering the area close to the user with a high LOD. Regions of the terrain further and further from the user would be rendered with less and less detail. The scene is dynamically updated each frame as the user moves around in the terrain environment. An often used algorithm for continuous LOD rendering in game engines is the Garland-Heckbert algorithm [Garland and Heckbert, 1997].

5.1.6 Terrain

Many games and simulators are based in an outdoor environment. For example flight simulators test the operator's skills at flying airplanes or jets. The missions are based on accomplishing goals such destroying other planes or bombing various targets. During each mission the planes is flying over terrain, whether sea or land, and it is important that the terrain look realistic. In such cases the extend of the world can be quit large and requires significant amount of modeling. Moreover, at run time this data needs to be effectively managed. A terrain system in a game engine has the job of supporting both the modeling process and run-time management. Terrain data is typically represented as height values sampled on either a rectangular lattice or a network of triangles. In either case the data sets tend to be large and make it difficult to render at real time rates for two reasons. First, the terrain data cannot fit entirely in memory, so it needs to be loaded from disk. Second, the renderer must process a large number of small triangles corresponding to distant terrain.

A way to overcome these difficulties is to use triangle reduction, where the terrain is divided into multi-resolution models. One possibility is to use discrete LOD, where the entire current resolution model is switched to a different resolution model based on the distance from the eye point. The problem with this approach is that the switching could be noticeable. A better approach would be to reduce the triangles in a way to minimize visual impact. This could be done be using continuous LOD, where the model is changed by a small number of triangles at a time. The idea is that two triangles are reduced to one triangle if the height variation between the two triangles is smaller that an specified number of pixels. Peter Lindstrom [Lindstrom et al., 1996] has developed an algorithm that uses this strategy. The algorithm consists of three phases: a coarse-level simplification based on blocks of the terrain, a fine-level simplification at the vertices within each block, and a rendering of each block.

5.2 Comparing Models Using Multiple Views

Validating graphical correctness of a 3D scene by capturing a single image of the objects appearing in the scene and comparing with a stored reference image is not considered the best approach, because a considerable amount of information is lost by the projection from 3D world coordinates to 2D image coordinates. Ideally, capturing the set of all radiance samples that emanate from the surface of an object under all possible lightning conditions would be desirable. This is obviously not possible in practice. A way to achieve an approximate 3D effect is to capture a collection of radiance samples of the scene from different camera positions, and applying an image metric to the entire set of captured images. This approach has been investigated by several authors in the field of computer graphics, where it has been used for different purposes. Peter Lindstrom has applied the principle of multiple camera views to simplify complex polygonal surfaces and optimizing their approximations. This work is described in his Ph.D. thesis [Lindstrom, 2000] and summarized in the article [Lindstrom and Turk, 2000]. Additionally, comparing 3D models using multiple views combined with image metrics has been studied in [Alface et al., 2005]. Here the authors use multiple views and image metrics, based on human visual perception, to create quality measurements for benchmarking 3D watermarking schemes.

In [5] the authors present an approach that matches 3D models using their visual similarities, which are measured with image differences in light fields.

A different approach to matching 3D models is taken in [Laga et al., 2004]. Here the authors brake with traditional methods where multiple 2D views of the same object are required to capture the relevant geometry features. Instead, they propose a technique that uses spherical parametrization and geometry images for 3D shape matching. This technique reduces the problem to 2D space without computing 2D projections and keeps the preservation of small details.

Although the purpose of rendering images from different camera positions in the articles mentioned above is different from the work described in this thesis, the method is still useful in the context of validating correctness of 3D computer graphics. Figure 5.3 shows a front-view image of a 3D model of a rabbit. In this example the camera position and orientation only shows a limited number of the polygons that constitute the 3D rabbit model. Suppose for the sake of argument, that the graphical correctness of the rabbit model needs to be verified with respect to the human visual perception of a rabbit. Only considering a single rendered image will be insufficient to determine whether the model is correct, because an image captured from a single camera position cannot succeed in expressing the full geometry and texture of the model. Based on the front-view image shown in figure 5.3, it would be impossible to establish whether the rabbit is entirely correctly represented. It seems correct, but nothing can be said about the parts invisible from this particular viewpoint.

A better approach would be to collect multiple images captured from different positions and orientations of the camera. Each of the collected images should be stored in a set together with a key specifying the position and rotation parameters for the particular image. To determine the graphical similarity between two 3D models, two individual image sets should be created. Each entry in one set should contain an image with identical camera parameters as the corresponding entry in the other set. By iterating through each of the sets and applying an image metric to each image pair it should be possible to achieve

Figure 5.3: An front-view image of a 3D model of a rabbit. The position and orientation of the camera only captures the front of the model.

an overall quantitative measure of the similarity between the two models. Figure 5.4 shows an image of the same 3D rabbit model as shown in figure 5.3. Here 12 images are collected from 12 different camera positions. This approach gives a much richer data collection which means that a more precise correctness evaluation can be performed.

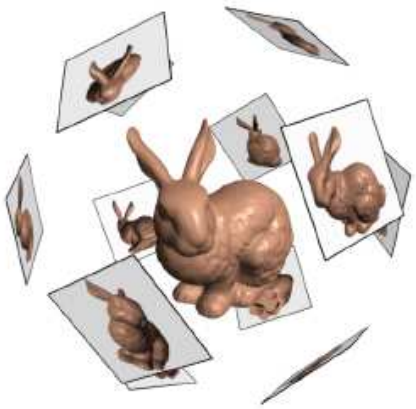


Figure 5.4: Example of a 3D model of a rabbit seen from 12 camera different camera positions. The viewpoints and view orientations are distributed uniformly.

5.2.1 Rendering Parameters

In order to compare 3D models using multiple scene views, there are a number of choices that need to made to collect and produce the images used to estimate the degree of similarity. Among these are image dimensions and camera settings for each image, the choice of background to render the objects on, light source information, and shading parameters. For most of these parameters, there is no obvious best choice. Therefore these parameters need to be chosen and

adjusted according to what would seem desirable and reasonable useful seen from a game/simulation development perspective. The following section describes the rendering parameters in the context of the virtual environments found in games and simulators.

Camera and Image Parameters

While the number of views required and the "optimal" placement of viewpoints vary between objects and scenes, the focus has been on using a fixed set of viewpoints, and require that they are distributed uniformly in direction.

To ensure even coverage of image samples, the viewpoints need to be arranged so they approximate a sphere of camera positions surrounding the object under consideration. Additionally, it would be desirable if the viewpoints are equidistant from each other. In practice, only five possible configurations exist for which the viewpoints are uniformly distributed. These are the vertices of the Platonic solids, which are discussed in more detail in section 5.4.1. Other reasonable configurations can be obtained using subdivision of these polyhedra, or by using the vertices of the semi-regular Archimedean solids. The work of Peter Lindstrom [Lindstrom, 2000] and [Lindstrom and Turk, 2000] uses the vertices of a Small Rhombicuboctahedron to position the camera as shown in figure 5.5.

Figure 5.5: The Small Rhombicuboctahedron. The 24 vertices of this uniform polyhedron are used as the viewpoints surrounding an object in the off-line quality evaluation in the work carried out by Peter Lindstrom.

The Small Rhombicuboctahedron is the sixth of the thirteen Archimedean solids. This particular solid has a total of 24 vertices [6], which gives the same number of camera positions. Using this solid would obviously give a more detailed image representation of the objects in the scene compared to for instance a Hexahedron (Cube), which has 8 vertices. The cost of a more detailed image representation is the computation time needed to calculate the image metrics. In actual high-quality graphics applications with screen resolution around 1024 $\times$ 768 or above, the captured images get the same size as the screen resolution. The time complexity needed to evaluate the previously mentioned image metrics is around $O(nm)$, where n is the height and m is the width. For 24 images, this might take a considerable amount of time, which would be critical if the

computations need to be performed in real-time - that is in parallel with the graphics application under consideration.

Another issue that needs to be addressed, is to ensure that the model that needs to be validated is entirely contained in each image. This can be done by calculating the minimum bounding sphere of its vertices. This sphere can be computed rather quickly by using the appropriate bounding sphere algorithm contained in Delta3D. Given a set of fixed camera positions, the camera at each viewpoint should be directed towards the center of the bounding sphere. This setup guarantees that two models are compared using identical camera positions and orientations.

Scene Background

Without any specific information about the environment in which the models will appear, predictions about scene background become virtually impossible. Model comparison would be significantly simplified if the scene background has a solid uniform intensity. Additionally, model comparison would be even more simplified if it could be guaranteed that no parts of the surface silhouette blend with the background.

Unfortunately, none of the above reasons can be guaranteed to hold in realistic 3D graphical application. In actual 3D virtual environments like games and simulators, some of the elements contained in a scene are represented by static meshes. Often such meshes constitute a major part of the map architecture used, like buildings and terrain elements contained in the virtual world. The word "static" refers to the fact that these meshes are not vertex animated. Unlike static meshes, animated entities in a scene use rigid-body dynamics or key frame animation to create a sense of motion, where the polygon vertices in the mesh change position and orientation over time. In graphical regression testing, verification of the static parts in a scene can be done directly in the modelling tool used to create the static meshes by inspecting them independently. For instance by observing the raw skeletal mesh before adding textures and later with textures added to the model. Testing the animated parts in an application, is more difficult as these often change their visual representation at certain stages during the game or simulation.

By capturing multiple scene views of a 3D application containing both static and animated meshes and storing these as bitmap images, which makes it very difficult to distinguish between static and animated meshes. This means that the scene background to a large extent is composed of whatever textures the static meshes use. The animated parts of the scene therefore become mixed with the background textures, making image segmentation very hard. Therefore image segmentation using region growing and blob detection algorithms [Sonka et al., 2007] and [Jain, 1989], is unlikely to be successful. Instead a simpler approach using different image metrics is considered.

5.3 Scene Graphs

This section gives a brief overview of scene graphs in general and how scene graphs is used in 3D graphics. Additionally, a few scene graph implementations are described.

Scene graphs help process large amounts of information by providing a logical model for sorting and identifying data. The model allows filtering of the data before it is sent to be rendered or executed, ensuring that only data which has a noticeable effect on the scene is processed. For example, a ball that is hidden entirely behind a wall will not be processed, since it cannot be seen and does not affect how the scene appears. Filtering the data saves both time an computational resources, helping in displaying the scene smoothly. A scene is a method of organizing the data that describes the geometry of a 3D computer generated scene.

5.3.1 Scene Composition And Management

Scene graphs address problems that generally arise in scene composition and management. There exist many scene graph implementations, but a common feature for these are that they shield the developer from the details of rendering, allowing the developer to focus on what to render, rather than how to render it.

As figure 5.6 illustrates, scene graphs offer a high level alternative to low level graphics rendering APIs such as OpenGL and Direct3D. In turn, they provide an abstraction layer to graphics subsystems responsible for processing and presenting screen data to the user. By separating the scene from the operations

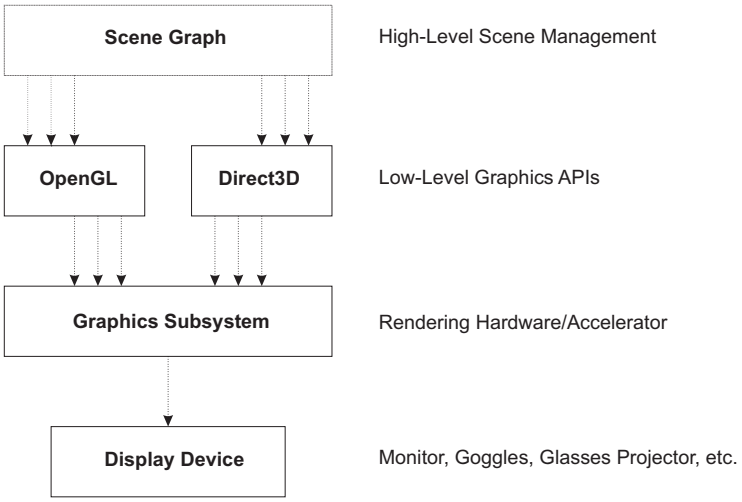


Figure 5.6: Scene graph programming models hides underlying graphics APIs and graphics rendering and display devices from the developer.

performed on it, the scene graph programming model establishes a clear boundary between scene representation and rendering. Thus, scenes can be composed and maintained independent of routines that operate on them.

5.3.2 Representation Of Scene Graphs

As figure 5.7 shows, scene graphs consist of nodes - that represents the objects in a scene connected by arcs - edges that define the relationships between nodes. Together, nodes and arcs produce a graph structure that organizes a collection of objects hierarchically, according to their spatial position in the scene. With

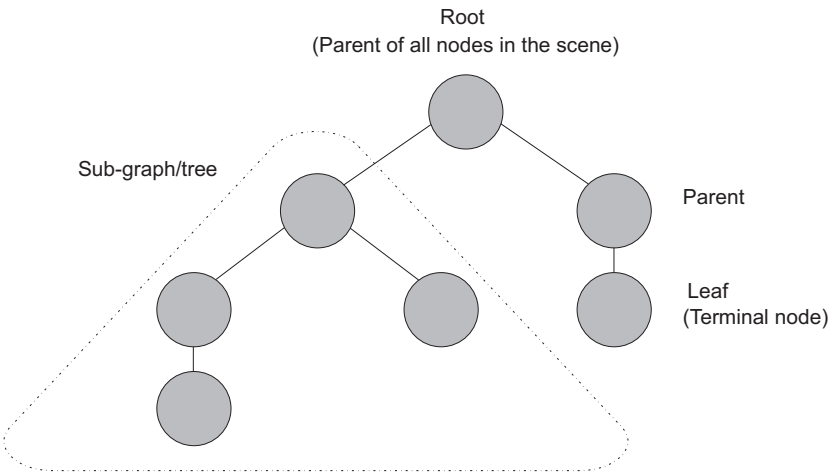


Figure 5.7: The structure of a scene graph represented as a tree.

the exception of the topmost root node, which defines the entry point into the scene graph, every node in a scene has a parent. Nodes containing other nodes are parent nodes, while the nodes they contain are the child nodes (children) of their parent. Nodes that can contain children are grouping nodes; those that cannot are leaf nodes. It is this distinction between leaf nodes and group nodes, which makes a scene graph represented as a tree different from a traditional search tree. Sub-graph structures let a specific grouping of nodes exist as a discrete and independently addressed unit of data within the main scene graph structure. Operations on the scene can be performed on all nodes in the graph, or they may be restricted to a particular sub-graph. Scenes can therefore be composed of individual nodes as well as entire sub-graphs that may be attached or detached as needed.

Scene graphs describe relationships between objects, and are used to pass information and effects along the graph based on these relationships, which reduces the amount of programming required. If an object that resides at a node $n1$ is affected by something, everything that is below $n1$ in the graph, that is any node that can be reached following a path away from $n1$, is also affected. By using relationships to propagate effects, the scene graph can be used to handle low level graphics code rather that a programmer coding the individually required updates and interactions for each affected node. Grouping related objects allows a program to apply change to an entire group of nodes rather that individually changing each node.

Scene graphs are often structured as a collection of nodes or vertices's in a directed acyclic graph (DAG). Nodes in a DAG are connected by unidirectional, that is one-way paths. In an acyclic graph, if one follow a path away from a node there is no way to return to that node by any path. In other words, no path through a DAG visits any individual node more that once. Represented as a DAG, the scene graph contains no cycles and nodes are allowed to have multiple children and parents. Another way the relationship between objects in the scene can be represented, is by a tree structure. Represented as a tree structure nodes in the scene graph have a single unique identifiable parent.

Both approaches have advantage and disadvantages, and it seems that both approaches are used in open source engines as well as engines delivered by commercial software vendors.

5.3.3 Open Scene Graph

The Open Scene Graph (OSG) is an open source, cross platform graphics toolkit for the development of 3D graphics applications such as flight simulators, games, virtual reality and scientific visualization. Based around the concept of a scene graph, it provides an object oriented framework on top of OpenGL freeing the developer from implementing and optimizing low level graphics calls.

OSG is designed for portability and scalability [Martz, 2007]. As a result it is useful on a wide variety of platforms, and renders efficiently on a large number of graphics hardware. OSG is designed to be both flexible and extensible, to allow adaptive development over time. To enable these design criteria, OSG is built with the following concepts and tools: ANSI C++, C++ Standard Template Library, and Design Patterns.

5.3.4 Scene Graph Matching

During the preliminary study, it was discussed whether scene graphs could be used to validate virtual environments. The idea was to store scene graphs as reference models instead of images, and perform scene graph matching by traversing the nodes in each graph and compare against the corresponding reference nodes. The strategy was that by serializing scene graphs, where the graphics had been validated, more information would be kept compared to image matching which does not contain information about the 3D effects contained in the scene environment.

However, investigations, carried out during the work performed in this thesis, have shown that implementing the above considerations into a framework would cause problems with the way OSG and Delta3D handle scene graphs. OSG supports scene graph specification in plain ASCII text format, but unfortunately most real virtual environment applications use 3D models developed by artists in proprietary or closed formats. Therefore, it was decided not to pursue scene graph matching further and instead focus on image matching.

5.4 Image Collector

This section describes the implementation of the image collector responsible of collecting images from different viewpoint. Initially, the incorporation of Platonic solids into multiple scene views is presented and the mathematics behind the implementation is explained. Subsequently, the interface between the image collector and Delta3D is described, including how to generalise the module in order to make it sufficiently abstract to be used in most Delta3D applications. Finally, a prototype application developed in Delta3D is presented and the steps necessary for using the image collector module in this prototype is described.

5.4.1 Multiple Views and Platonic Solids

As previously described the Platonic solids have some properties which make them suitable for realizing a module to collect multiple scene views in a 3D virtual environment. The Platonic solids, also called the regular polyhedra, are convex regular polygons. There are exactly five such solids: the tetrahedron, hexahedron (cube), octahedron, icosahedron, and dodecahedron [7]. These solids are interesting in the context of computer graphics, because they can be specified by two parameters, the edge length a and the center point $c = (x_c, y_c, z_c)$. By using these two parameters it is possible to calculate the coordinates of all the vertices that constitute the particular solid. The idea is to associate a Delta3D camera of type `dtCore::Camera` to a Platonic solid and rotate this camera around the circumscribed sphere of the solid. At each vertex a scene view is taken and stored as a bitmap image. The orientation of the camera is directed towards the center point c, which ensures that the graphical object being regression tested is in the view frustum of the camera and therefore stored in the bitmap image.

The implementation of the image collector module includes different Platonic solids, which are used to surround scene objects in order to capture multiple screenshots of these. A detailed derivation of the equations describing the vertex positions of the solids is found in appendix C.

5.4.2 Interface to Delta3D

In order to design the image collector as a module sufficiently abstract to provide functionality for most graphical applications implemented in Delta3D, the structure of a number of Delta3D applications have been investigated. The purpose of this analysis has been to find an interface in Delta3D to which the image collector could be attached.

The Delta3D source code provides a number of application examples that show how the classes are used to implement typical game and simulation features. These examples vary from showing how a simple "Hello Wold" text could be rendered, to more advanced features such as AI (artificial intelligence), collision detection, lightning and shading parameters, network communication, etc. A common component in all Delta3D applications that displays and renders 3D models, is the camera which is responsible of showing the scene objects. The camera class in Delta3D is implemented in the class `dtCore::Camera`. Normally, a Delta3D application uses a single camera, which renders the entire scene by using different motion models. Currently, Delta3D supports various motion models, where `dtCore::MotionModel` constitutes the base class for all motion models. The following shows the concrete motion models available to the programmer.

- Collision motion model - `dtCore::CollisionMotionModel` - uses the Open Dynamics Engine (ODE) collision meshes to allow typical first person shooter (FPS) camera interaction with the environment.
- Fly motion model - `dtCore::FlyMotionModel` - simulates the action of flying.
- FPS motion model - `dtCore::FPSMotionModel` - used for typical first person shooter motion.

- Orbit motion model - `dtCore::OrbitMotionModel` - causes its target to orbit around a point.
- RTS motion model - `dtCore::RTSMotionModel` - controls the camera used in typical real time strategy (RTS) motion.
- UFO motion model - `dtCore::UFOMotionModel` - simulates the action of flying in a UFO.
- Walk motion model - `dtCore::WalkMotionModel` - simulates the action of walking or driving.

Although, most Delta3D applications use one or more motion models, each motion model is attached to the default camera available through the application class - `dtABC::Application`. This class is the generic base level class for most applications. It contains the basic components required for applications developed within the Delta3D framework. Instantiating an object of this class, creates access to internal attributes, like the window found in `dtCore::DeltaWin` which specifies the rendering area that the camera renders to, the camera found in `dtCore::Camera` which is the default camera, and the scene `dtCore::Scene` which encapsulates the root of the scene graph.

To avoid making the image collector unnecessarily tightly coupled to the Delta3D application which should be regression tested, a solution could be to let the image collector control its own camera. This would ensure that multiple scene views could be collected without interfering with the application camera, and thereby avoid having the application code and the image collector code interlinked.

Delta3D however, does not allow two or more cameras to be enabled and active at the same time if the cameras render to the same window. A way to avoid this problem could be to let the camera in the image collector render to a window separate from the window used by the Delta3D application. Unfortunately, this approach causes synchronization problems between the different windows. Additionally, programmers might prefer to be able to switch the image collector module on and off directly in the application at runtime.

Letting the image collector have its own camera is not sufficient to avoid the "one active camera per window" restriction, because the image collector has no way to enable or disable the application's camera when needed. As mentioned above, a Delta3D application inherits from `dtABC::Application`, which makes it possible to get access to the default camera. Therefore, a solution would be to attach a `dtABC::Application` object to the image collector. This could be done as a member variable, and thereby using a composition strategy or it could be done through inheritance making the image collector a subclass of `dtABC::Application`. However, the latter approach can be discussed since the image collector can not really be considered a Delta3D application. Instead, the image collector is responsible of bridging the gap between the application under regression test and the rest of the test tool. The image collector is considered a stand-alone application, which uses the class `dtABC::Application` as a link to gain access to internal Delta3D variables holding the application camera and the application scene. Therefore, a `dtABC::Application` object is given as a pointer to the constructor responsible of setting up the multiple views and thereby the image collector. The exact implementation details are discussed in section 5.4.3.

5.4.3 Structure of the Image Collector

In the following the classes contained in the image collector module are described and the responsibility of each class is explained. Figure 5.8 shows the class diagram of the image collector component. As shown, the image collector is

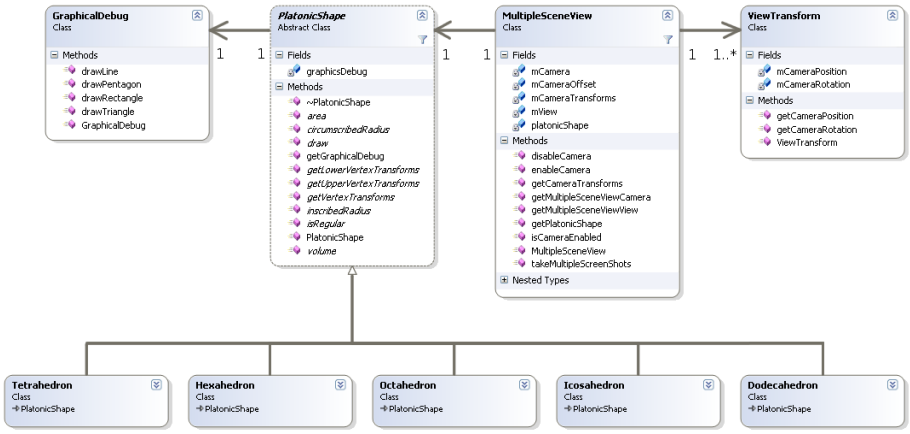


Figure 5.8: Class diagram showing the structure of the Image Collector. Only the most important attributes and methods are shown.

composed of the following classes:

- `PlatonicShape` - abstract class providing a general Platonic shape.
- `Tetrahedron` - represents a tetrahedron shape.
- `Hexahedron` - represents a hexahedron shape.
- `Octahedron` - represents an octahedron shape.
- `Icosahedron` - represents an icosahedron shape.
- `Dodecahedron` - represents a dodecahedron shape.
- `GraphicalDebug` - provides different drawing methods primarily useful when debugging graphical virtual environment applications.
- `ViewTransform` - represents a data structure used to hold camera transformations.
- `MultipleSceneView` - provides a coupling between the application under regression test and the test tool. This is the class users instantiate in order to use the image collector.

For clarity the field attributes and member variables shown in the class diagram are only a subset of the members contained in the classes. In the following a brief description of the classes contained in the image collector module is given.

Class PlatonicShape

The class `PlatonicShape` is an abstract base class responsible of implementing a general interface to the five Platonic solids described in section 5.4.1. The main purpose of this class is to specify the functionality needed to place cameras at the vertices of a given Platonic solid in a virtual Delta3D environment. The actual camera placement is handled by the constructor of the class `MultipleSceneView`, which is describe later. In order to extract the camera position and rotation of a given solid, three pure virtual methods are specified. Making these methods pure virtual ensures that the subclasses, which implement the concrete Platonic solids, provide an implementation of these methods. The three methods are:

- `getLowerVertexTransforms` - returns a vector containing the positions and rotations of the vertices below the center point of a given solid.
- `getUpperVertexTransforms` - returns a vector containing the positions and rotations of the vertices above the center point of a given solid.
- `getVertexTransforms` - returns a vector containing all the positions and rotations of the vertices of a given solid.

The reason for not always returning all vertex transforms of a given solid, is that in some situations the virtual objects encapsulated by the solid are placed on a landscape plane in the virtual world. Consider for instance a character walking on a planar surface in a virtual environment, then the ground constitutes a surface below the character. Taking multiple scene views below the center point of the character could result in views showing only the ground surface seen from below. In such situations differentiation between which vertices that are included in the view becomes relevant.

Additionally, the class allows the programmers to visualize the Platonic shapes in the scene, by supplying different drawing methods. The drawing methods are included to make debugging easier in the sense that placing the Platonic solids in a scene without being able to visualize these is considered problematic, since there is no way to guarantee that a solid is not placed inside a wall or interfere with another object in the scene. This could be avoided by implementing collision detection of the solids. Instead the drawing methods make it possible to visually determine if a solid collide with scene objects.

Besides the methods discussed above, the `PlatonicShape` class contains functionality to perform a number of geometry calculations. These include finding area, volume, inscribed radius and circumscribed radius of a given solid. These methods are all pure virtual and the implementations are done in the appropriate subclasses.

Class GraphicalDebug

The class `GraphicalDebug` is a drawing utility class, which provides simple drawing functionalities. The class contains methods to draw simple two-dimensional geometries in a three-dimensional Delta3D virtual environment. The geometries include: lines, triangles, rectangles, and pentagons. These geometries are all returned as pointers of type `osg::Geometry`, which means that it is up to the

Delta3D application programmer to add the geometries to the scene graph, if they should be rendered and displayed by the application.

Besides providing basic drawing capabilities, which might be considered useful in isolation, the `GraphicalDebug` class is used by the `PlatonicShape` class and therefore by its subclasses. Each of the Platonic solids provide an implementation of the `draw` method, which obviously draws the particular solid in the same manner as the more simpler two-dimensional geometries. Since all the Platonic solids are composed of triangles, rectangles, or pentagons, the draw methods in each of the subclasses use the simpler two-dimensional geometries to draw the particular solid.

It could be argued that the `GraphicalDebug` class should be a nested class of `PlatonicShape`. However, it was considered useful to allow public access to the simpler two-dimensional drawing methods contained in the `GraphicalDebug` class. The line drawing method might be useful as a simple graphical debug utility, for example to establish whether or not there exists line of sight between two points in a virtual environment without having to perform more advanced collision calculations.

Class ViewTransform

The class `ViewTransform` provides a simple data structure used by the other classes. It simply encapsulates two Open Scene Graph objects of type `osg::Vec3` into a single object, which holds the camera position and the camera orientation as three-dimensional vectors. This is useful when multiple scene views have to be created and later fetched from an Delta3D application. In this situation dynamic data structures, like vectors and maps from the standard template library, can be used as collections holding these transformation objects. More details about creating and using dynamic data structures in C++ can be found in [Meyers, 2001] and [Josuttis, 1999].

Class MultipleSceneView

The class `MultipleSceneView` has a direct interface to Delta3D and is the class that users of the image collector module interacts with in order to setup multiple scene views in an Delta3D application. The constructor takes a pointer of type `dtABC::Application` which makes it possible to get references to internal objects like camera, scene, view, etc. in the Delta3D application under test.

An extract of the source code implementation contained in the constructor of the class `MultipleSceneView` is shown in listing 5.1. Besides the input pointer to the Delta3D application, a view type parameter specifying what type of multiple scene view that should generated, is given as input. The type of scene view corresponds to the vertex transforms in one of the five Platonic solids. The third and fourth parameters specify the center point and the size of the multiple scene view respectively. Instantiating an `MultipleSceneView` object creates a local camera and a local scene view with reference to a Delta3D application. Upon creation, the local view, contained as a member of the `MultipleSceneView` object, is added to the scene graph of the application through the parameter pointer. Initially, the local camera is disabled which ensures that it does not interfere with the default application camera. A `MultipleSceneView` object has a member variable of type `PlatonicShape`, which is used as a polymorphic

```
MultipleSceneView::MultipleSceneView(dtABC::Application *application,
                                     SceneView viewType,
                                     const osg::Vec3& center,
                                     const float edgeLength)
{
  mCamera = new dtCore::Camera();
  mView = new dtCore::View();
  mView->SetScene(application->GetScene());
  application->AddView(*mView);
  mCamera->SetWindow(application->GetWindow());
  mCamera->SetEnabled(false);
  mView->SetCamera(mCamera.get());
  mCamera->GetOSGCamera()->setAllowEventFocus(false);

  switch(viewType)
  {
    case TetrahedronView:
    {
      platonicShape = new Tetrahedron(center, edgeLength);
      mCameraTransforms = platonicShape->getVertexTransforms();
      break;
    }
    case TetrahedronUpperView:
    {
      platonicShape = new Tetrahedron(center, edgeLength);
      mCameraTransforms = platonicShape->getUpperVertexTransforms();
      break;
    }
    case TetrahedronLowerView:
    {
      platonicShape = new Tetrahedron(center, edgeLength);
      mCameraTransforms = platonicShape->getLowerVertexTransforms();
      break;
    }
    case HexahedronView:
    {
      platonicShape = new Hexahedron(center, edgeLength);
      mCameraTransforms = platonicShape->getVertexTransforms();
      break;
    }
    .
    .
    .
  }
}
```

Listing 5.1: Extract of the source code in the `MultipleSceneView` constructor.

variable controlling which vertex transforms should be used to create the multiple scene view. This member variable is assigned to one of the five Platonic solids dependent on the view type given as input.

Listing 5.2 shows the implementation of the method used to generate multiple screen captures and store these as bitmap images. This method is a member of the `MultipleSceneView` class. The images are generated by iterating through the vertices of a given Platonic solid and for each vertex the camera position and rotation coordinates are extracted and passed to the local camera, where the Delta3D method `dtCore::Camera::TakeScreenShot` is called to generate the bitmap images. The images are given filenames corresponding to unique timestamps of their creation.

5.4.4 Using the Image Collector in a Delta3D Application

The following section describes how the image collector module could be used in a Delta3D application. First a presentation of the application used to experiment with the image collector module is given. Finally, the necessary steps needed in order to use the image collector in the prototype are described.

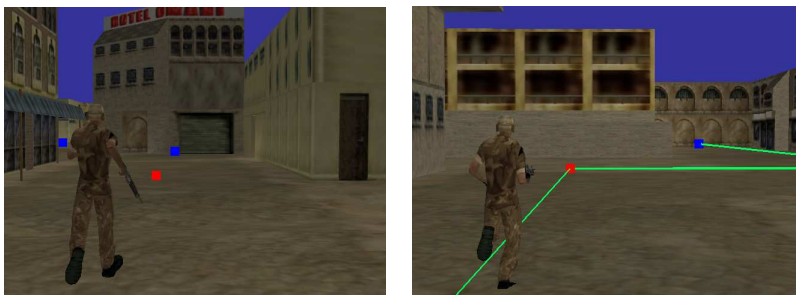
Walking Soldier Application

The application used as case scenario for graphical testing, is an extension of the prototype developed in the preliminary study of this thesis. More details about

```
void MultipleSceneView::takeMultipleScreenShots(dtABC::Application *application)
{
   if(application->GetCamera()->GetEnabled())
   {
      application->GetCamera()->SetEnabled(false);
      mCamera->SetEnabled(true);
      dtCore::Transform xform;
      for(std::vector<ViewTransform>::const_iterator it = mCameraTransforms.begin();
      it < mCameraTransforms.end(); it++)
      {
         std::string namePrefix;
         std::stringstream out;
         ViewTransform currentViewTransform = *it;
         osg::Vec3 cameraPosition = currentViewTransform.getCameraPosition();
         osg::Vec3 cameraRotation = currentViewTransform.getCameraRotation();
         xform.SetTranslation(cameraPosition.x(), cameraPosition.y(), cameraPosition.z());
         xform.SetRotation(cameraRotation.x(), cameraRotation.y(), cameraRotation.z());
         mCamera->SetTransform(xform);
         out << count;
         namePrefix = out.str();
         mCamera->TakeScreenShot("screenshots/" + namePrefix);
         count++;
         dtCore::System::GetInstance().StepWindow();
      }
   }
   mCamera->SetEnabled(false);
   application->GetCamera()->SetEnabled(true);
}
```

Listing 5.2: Method `takeMultipleScreenShots` takes screenshots from multiple viewpoints .

this initial work can be found in [Horn and Grønbæk, 2008]. The application consists of an animated character, modeled as a soldier. The soldier navigates through a town following a specific path. The graphical scene is a 3D model, composed of different static objects like houses, rocks, trees, walls, etc. - all with textures providing a more realistic environment. Figure 5.9 shows the animated soldier navigating around in the town environment. The red and blue cubes in (a) represent waypoints, which are used to create nodes in the navigation graph. In (b) the waypoints are connected with lines, which represent edges in the navigation graph. The navigation graph is used to calculate valid paths that the soldier can follow.



(a) (b)

Figure 5.9: Animated soldier in town. (a) Normal view. (b) Navigation edges shown.

The route that the soldier follows, is created by inserting waypoints directly into the map of the town. This is done in Delta3D's level editor, STAGE. Insertion of waypoints is accomplished via a manual graphical editing process, where the waypoints are visualized as simple billboards. When editing the level, the billboards are located just as any other objects (e.g. tree or rock)

in the scene. Upon saving the level, a ray is traced between nearby pairs of waypoints, and if the ray does not intersect anything, a directed edge is added to the navigation graph. The waypoints are saved in a file that can be parsed by methods from the Delta3D API.

Figure 5.10 shows the town seen from a camera position above the buildings. The green rays shown in (b), represent the navigation edges in the navigation graph. By following the navigation edges, the soldier is able to move around without colliding with any obstacles present in the environment.

(a) (b)

Figure 5.10: Town model. (a) seen from a top camera position. (b) Navigation edges shown.

The application consists of the following source files:

- `AICharacter.h` - provides a class interface to the implementation of the AI character.

- `AICharacter.cpp` - implements the behavior of the soldier character. This class is responsible for the logic found in the implementation. That is the basic AI like path planning methods. The route calculation is performed by Delta3D's path planning algorithm. At runtime, the API provides a method that will attempt to plan a route between two points in the level. This is accomplished by searching the navigation graph using the A* (A-star) algorithm. The function returns the shortest path to the goal location in the form of a list of waypoints. The actual use of the waypoints to move the character is not handled by Delta3D, but is handled by user code - in this case by the `AICharacter` class. Listing 5.3 shows how this is done in the WalkingSoldier application. The method `FindPathAndGoToWaypoint` is member of class `AICharacter`. This method determines if the exists a path between the current waypoint and another waypoint specified as input parameter. The call to Delta3D's A* algorithm, returns whether a path is found or not. If a path is found, the body of the `if`-statement is executed and every waypoint is rendered green, except the last which is rendered red. If no path is found `FindPathAndGoToWaypoint` returns false.

- `TestAI.h` - provides a class interface for testing the application.

- `TestAI.cpp` - provides a test of the AI character and is responsible of setting up the scene, placing the camera, loading the map, and loading the waypoints. The class `TestAI` extends the class `Application`, which is part of the Delta3D engine. The `Application` class contains a template for creating an application in Delta3D. This template consists of a number of virtual methods, that provide a basic skeleton for an application. The class `TestAI` overrides the following virtual methods from the `Application` class:
  - `Config`: Sets up the scene by placing the camera and the loading graphics model and places the character at the first waypoint.
  - `KeyPressed`: Makes the user interact with the application by using the keyboard.
  - `PreFrame`: Called by the application before drawing the scene and called in the application loop.
- `Main.cpp` - responsible of starting the application.

```
bool AICharacter::FindPathAndGoToWaypoint(const Waypoint* pWaypoint)
{
   mAStar.Reset(mCurrentWaypoint, pWaypoint);
   PathFindResult pHasPath = mAStar.FindPath();

   if(pHasPath != NO_PATH)
   {
      mWaypointPath = mAStar.GetPath();
      for_each(mWaypointPath.begin(), mWaypointPath.end(), funcRenderGreen());
      pWaypoint->SetRenderFlag(Waypoint::RENDER_RED);
      return true;
   }
   return false;
}
```

Listing 5.3: Method `FindPathAndGoToWaypoint` uses the A* path finding algorithm to calculate a path between the waypoints.

Multiple Scene Views in the Walking Soldier Application

The following section explains how the image collector is used in an Delta3D application. Having introduced the WalkingSoldier soldier application in the previous section, the following section presents how the image collector is used in this application.

Suppose that a programmer is interested in observing the soldier at carefully selected places in the virtual environment. It could be that the logical model behind the soldier made him change visual appearance at certain stages during the simulation. One could imagine that his inventory should change at these stages, causing his visual representation to undergo a transformation. Maybe he should carry a weapon at certain phases and at others the weapon should be concealed or entirely hidden. The image collector component could be used in such a scenario. The class `MultipleSceneView` is used to setup multiple scene views at these critical stages. Listing 5.4 shows an example of how the code to setup multiple views at the waypoints in the Walking Soldier application look like. This is simply done by iterating through the waypoints and for each way point a multiple scene view is added to a vector. The vector storing the multiple

```
void TestAI::SetupMultipleSceneView(MultipleSceneView::SceneView
viewType, const float edgeLength)
{
   const WaypointManager::WaypointMap& waypoints =
   WaypointManager::GetInstance().GetWaypoints();
   WaypointManager::WaypointMap::const_iterator begin = waypoints.begin();
   WaypointManager::WaypointMap::const_iterator end = waypoints.end();
   while(begin != end)
   {
      const Waypoint* aWaypoint = (*begin).second;
      const osg::Vec3 aWaypointPosition = aWaypoint->GetPosition();
      if(mDebug)
      {
         PrintSingleWaypoint(aWaypoint);
      }
      mSceneViews.push_back(MultipleSceneView(this, viewType,
      aWaypointPosition, edgeLength));
      ++begin;
   }
}
```

Listing 5.4: Method `SetupMultipleSceneView` creates multiple scene views at the waypoints in the Walking Soldier application.

scene views is kept as a member variable in the application under regression test, which makes it possible to access the views from other application methods.

The strategy used in 5.4, creates the same type of scene view at all the waypoints. If for instance hexahedron views should be used at certain waypoints and octahedron views should be used at other waypoints, these views could simply be created by calling the `MultipleSceneView` constructor with the appropriate waypoint coordinates and view types.

The image collector makes it possible to visualize the multiple scene views by drawing the Platonic shapes used. Listing 5.5 shows the code needed to visualise the scene views in the Walking Soldier application.

```
void TestAI::DrawPlatonics()
{
   std::vector<MultipleSceneView>::const_iterator begin = mSceneViews.begin();
   std::vector<MultipleSceneView>::const_iterator end = mSceneViews.end();
   while(begin != end)
   {
      MultipleSceneView currentSceneView = *begin;
      PlatonicShape *currentPlatonic = currentSceneView.getPlatonicShape();
      dtCore::Transformable *currentTransformable =
      currentPlatonic->draw(osg::Vec4(1, 1, 1, 1), 2.0f);
      GetScene()->AddDrawable(currentTransformable);
      ++begin;
   }
}
```

Listing 5.5: Method `DrawPlatonics` draws all the Platonic solids used in the scene views.

The application class `TestAI` also contains the method `GoToWaypoint`, which loops through the list of waypoints and sends the character to the waypoint specified by the input parameter. The `GoToWaypoint` method is shown in listing 5.6. The last `if`-statement checks whether frame capturing is enabled. If this is the case, multiple scene images are stored on disk each time the soldier arrives at a waypoint.

5.4.5 Testing the Image Collector

This section describes some test results of the image collector. Here results obtained using the image collector in the WalkingSoldier application are presented.

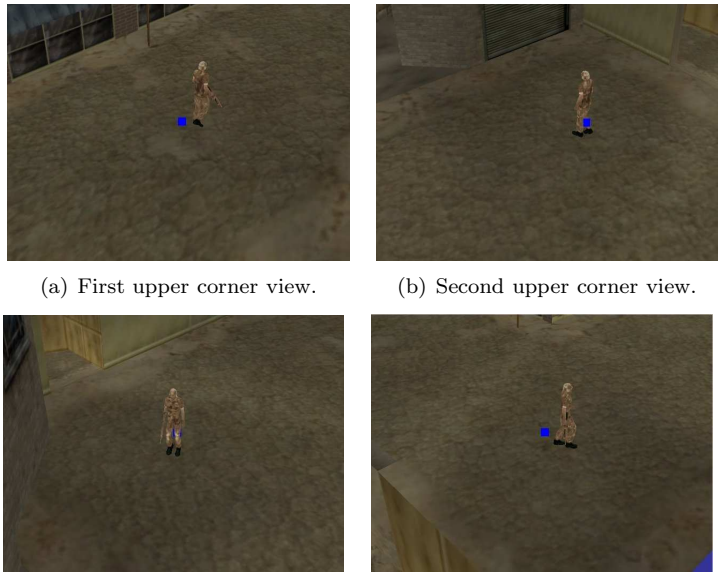
```
bool TestAI::GoToWaypoint(int pWaypointNum)
{
  const WaypointManager::WaypointMap& pWaypoints =
  WaypointManager::GetInstance().GetWaypoints();
  WaypointManager::WaypointMap::const_iterator iter = pWaypoints.begin();
  WaypointManager::WaypointMap::const_iterator endOfMap = pWaypoints.end();
  bool pHasPath = false;
  int i = 0;
  while(iter != endOfMap)
  {
    if(i == pWaypointNum)
    {
      pHasPath = mCharacter->FindPathAndGoToWaypoint((*iter).second);

      if(pHasPath)
      {
        mCurrentWaypoint = (*iter).second;
        MultipleSceneView currentView = GetSceneView(i-1);
        const std::vector<ViewTransform> camTransforms =
        currentView.getCameraTransforms();
        if(mDebug)
        {
          PrintSingleWaypoint(mCurrentWaypoint);
        }

        if(mEnableCapture)
        {
          currentView.takeMultipleScreenShots(this);
        }
        return true;
      }
      break;
    }
    ++i;
    ++iter;
  }
  return false;
}
```

Listing 5.6: Method `GoToWaypoint` sends the character to the waypoint specified by the input parameter. Additionally, the method calls the `takeMultipleScreenShot` method, which captures and stores a number of scene images.

The image collector has been tested on the WalkingSoldier application, where different multiple scene views were inserted and screen shots were captured from camera positions corresponding to the vertex transformations of different Platonic solids. The primary focus in the test, was to assert that the soldier was contained in the all the captured images. Figure 5.4.5 shows four images from the WalkingSoldier application. The images were captured by the image collector using the upper corner views of the hexahedron solid. As shown, the soldier is present in all four images and is approximately centered in the cube. The scene views shown above, depict the soldier at the same position in the virtual environment. The center of the hexahedron (not drawn), is placed at the exact location of the waypoint and the images are captured at the moment the soldier arrives at the waypoint. If the images are carefully studied, it can be notices that the soldier has changed position and rotation in the intervening time between each image. This occurs because the soldier moves while the images are captured and the files written to disc. However, this does not affect the result of the subsequent image matching, since the reference images were captured using the same strategy.



(a) First upper corner view. (b) Second upper corner view.

(c) Third upper corner view. (d) Fourth upper corner view.

Figure 5.11: Four different views of the soldier generated by the image collector.

Chapter 6

Image Regression Testing

This chapter describes how the collected scene images are processed in order to assist the tester/programmer in determining whether the regression test should pass or fail. Initially, some concepts in digital image processing, that we have used in the development of the graphical test tool for use in 3D virtual environments, is presented. Hereafter a brief and concise introduction to digital image representation is given, including how images are discretized in order to be processed. Next, different metrics used in image comparison are considered in the context of capturing multiple views of 3D scene objects. Hereafter, visual perception issues in image processing are described, including details on how images are perceived by the human visual system and how these effects can be incorporated into image metrics. Finally, the design, implementation, and test of the image analyzer are presented.

6.1 Basic Concepts

An image can be modeled by a continuous function of two or three variables; in the simple case arguments are coordinates (x, y) in a plane, while if images change in time a third variable t might be added. The image function values correspond to the brightness at image points.

The image on the human eye retina or on a computer monitor is intrinsically two-dimensional (2D). The real world which surrounds us is intrinsically three-dimensional (3D). The 2D intensity image is the result of a perspective projection of the 3D scene, which is modeled by a pin-hole camera.

In figure 6.1, the image plane has been reflected with respect to the xy plane in order not to get a mirrored image with negative coordinates; the quantities x, y, and z are coordinates of the point P in a 3D scene in world coordinates, and f is the focal length of the lens. The projected point has coordinates (x', y') in the 2D image plane, where the projected points are given by equation 6.1.

$$x' = \frac{xf}{z} \qquad y' = \frac{yf}{z}. \tag{6.1}$$

A non-linear perspective projection is often approximated by a linear parallel projection, where $f \to \infty$. Implicitly, $z \to \infty$ too - ortographic projection is a limiting case of perspective projection for faraway objects.

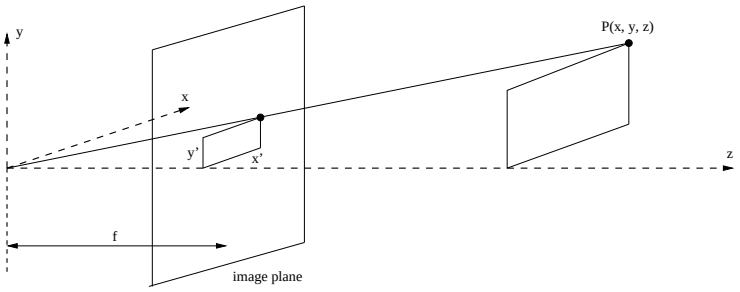


Figure 6.1: Geometry in perspective projection.

Image processing often deals with static images, in which time t is constant. A monochromatic static image is represented by a continuous function $f(x, y)$ whose arguments are two coordinates in the plane. Computerized image processing uses digital image functions which are usually represented by matrices, so coordinates are integer numbers. The domain of the image function is a region R in the plane given by equation 6.2.

$$R = \{(x, y), 1 \leq x \leq x_m, 1 \leq y \leq y_n\}, \tag{6.2}$$

where x_m and y_n represent maximal image coordinates. The image function has a limited domain - infinite summation or integration limits can be used, as it is assumed that the image function value is zero outside the domain R. The range of the image function values is also limited; by convention, in monochromatic images the lowest value corresponds to black and the highest white. Brightness values bounded by these limits are gray-levels.

6.2 Image Metrics

The image metrics used in the image analyzer are based on pixel differences in the collected scene images and statistical correlations between these. In the following a formal definition of image metrics is given including the notation used when describing image metric algorithms.

An image metric is a function over pairs of images that gives a non-negative measure of the distance between the two images. Roughly speaking, the larger the distance, the more dissimilar the images appear, while a zero difference implies that the two images are identical. In the following different image metrics are considered and evaluated within the context of this thesis.

Formally, a metric is a real-valued, non-negative function d between a pair of objects x and y that satisfies the following properties:

1. $d(x, y) = 0 \Longleftrightarrow x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z)$

According to [Lindstrom, 2000], few image metrics satisfy all three properties. The property most often violated is property 3 - called the "triangle inequality".

However, this violation is more of theoretical nature and does not seem to affect the way image metrics are applied in image algorithms.

6.2.1 Image Definitions And Notations

The following paragraph introduces some definitions and notations used in image processing and especially in the context of image comparison. In these definitions the pixel lattices of images A and B will be referred to as $A(i,j)$ and $B(i,j)$, where $i = 1,...,N$ and $j = 1,...,M$, since the lattices are assumed to have dimensions of $N \times M$. The pixels can take values from the set $\{0,...,G\}$ in any spectral band, where G is a value describing the number of colors in each band. The multi-spectral components of an image at positions i and j, and in band k is denoted $C_k(i,j)$, where $k = 1,...,K$. The boldface symbols $\boldsymbol{C}(i,j)$ and $\hat{\boldsymbol{C}}(i,j)$ indicate the multi-spectral pixel vectors at position (i,j) in the original image and the reference image respectively. For color images in the RGB representation the multi-spectral pixel vector at position (i,j) is given by $\boldsymbol{C}(i,j) = [R(i,j), G(i,j), B(i,j)]^T$.

- $C_k(i,j)$ the (i,j)th pixel of the kth band of image C.
- $\boldsymbol{C}(i,j)$ the (i,j)th multi-spectral (with K bands) pixel vector.
- $\boldsymbol{C}$ multi-spectral image.
- C_k the kth band of a multi-spectral image.
- $\epsilon_k = C_k - \hat{C}_k$ error over all pixels in the kth band of a multi-spectral image C.
- $\|\boldsymbol{C}(i,j) - \hat{\boldsymbol{C}}(i,j)\|^2 = \sum_{k=1}^{K}[C_k(i,j) - \hat{C}_k(i,j)]^2$ sum of errors in the spectral components at pixel position (i,j).

The notation used above is used in the following in order to express different image measures which are useful in implementing the image analyzer.

6.2.2 Measures Based On Pixel Differences

The digital image processing literature contains a wide variety of image metrics. A significant number of articles have been published about using image metrics as quality measures of distorted images. Although most of these metrics are used to compare compressed images with their non-compressed counterparts, some of these also apply to comparing visual similarity of images. The article [Avcıbaş et al., 2002] contains a comprehensive statistical evaluation of different image metrics. Here the authors categorize different image quality measures, extend measures defined for gray scale images to their multispectral case, and propose novel image quality measures. These measures are categorized into pixel difference based, correlation based, edge based, spectral based, context based, and human visual system (HVS) based measures. In the following different metrics based on the pixel difference between two images are presented.

Minkowsky Metrics

The L_γ norm of the dissimilarity of two images can be calculated by taking the Minkowsky average of pixel differences spatially and then chromatically - that is, over the K bands. The general Minkowsky metric is given by equation 6.3:

$$\epsilon^\gamma = \frac{1}{K}\sum_{k=1}^{K}\left\{\frac{1}{NM}\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}|C_k(i,j)-\hat{C}_k(i,j)|^\gamma\right\}^{1/\gamma}. \tag{6.3}$$

For $\gamma = 1$ the absolute difference metric is obtained - denoted D_1 in equation 6.4:

$$D_1 = \frac{1}{K}\frac{1}{NM}\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}\|\boldsymbol{C}(i,j)-\hat{\boldsymbol{C}}(i,j)\|. \tag{6.4}$$

For $\gamma = 2$ the mean square error metric is obtained - denoted D_2 in equation 6.5:

$$D_2 = \frac{1}{K}\frac{1}{NM}\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}\|\boldsymbol{C}(i,j)-\hat{\boldsymbol{C}}(i,j)\|^2. \tag{6.5}$$

The mean square metric gives a tolerable measure of image similarity, but often more advanced metrics are required, especially if human visual perception is a matching criteria. However, despite this, the mean square metric is used extensively in image matching due to its simplicity and straightforward implementation. Another reason for this is, according to [Avcıbaş et al., 2002], that the signal-to-noise ratio (SNR) often is defined in terms of the mean square metric

$$\text{SNR} = 10\log_{10}\frac{\sigma^2}{D_2}, \tag{6.6}$$

where σ^2 is the variance of the reference image. Another definition of the SNR is called the peak signal-to-noise ratio (PSNR), which is defined as

$$\text{PSNR} = 10\log_{10}\frac{(\text{peak-to-peak value of reference image})^2}{D_2}, \tag{6.7}$$

where the peak-to-peak value is the maximum possible pixel value in the image. These metrics are commonly used to quantify the degradation in the lossy compression of images [Ives et al., 1999] and [Sun and Li, 2005].

6.2.3 Correlation Based Measures

The process of determining a measure of the degree of similarity (or dissimilarity) between images, does to a large extent rely on statistical approaches in image matching. This means that techniques from statistics like correlation, covariance, and principal component analysis measures are often used to establish a quantitative measure of how much images are related. These measures describe the similarity between images, hence in this sense they are complementary to the difference measures discussed previously.

Structural Content Measure

The following presents some image matching measures, which use these concepts. The structural content measure is given by

$$C_1 = \frac{1}{K}\sum_{k=1}^{K}\frac{\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}C_k(i,j)^2}{\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}\hat{C}_k(i,j)^2}. \tag{6.8}$$

Cross-correlation Measure

Another often used statistical measure is the normalized cross-correlation metric

$$C_2 = \frac{1}{K}\sum_{k=1}^{K}\frac{\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}C_k(i,j)\hat{C}_k(i,j)}{\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}C_k(i,j)^2}. \tag{6.9}$$

Czenakowski Measure

A metric that is useful for comparing vectors with strictly non-negative components, like in the case of color images, is given by the Czenakowski distance

$$C_3 = \frac{1}{MN}\sum_{j=0}^{M-1}\sum_{i=0}^{N-1}\left(1 - \frac{2\sum_{k=1}^{K}\min[C_k(i,j),\hat{C}_k(i,j)]}{\sum_{k=1}^{K}[C_k(i,j)+\hat{C}_k(i,j)]}\right). \tag{6.10}$$

As the difference between two images C and $\hat{C}$ tends towards zero $\epsilon = C - \hat{C} \to 1$, all the correlation based measures tend towards one. Additionally, the distance measures and correlation measures are complementary, so that under certain conditions, minimizing distance measures is tantamount to maximizing the correlation measure.

Angle Moment Measure

A variant of correlation based measures can be obtained by considering the statistics of the angle between the pixel vectors of the original image and the reference image. Similar colors will result in vectors pointing in the same direction, while significantly different colors will point in different directions in $\boldsymbol{C}$ space. The vectors $\boldsymbol{C}$ and $\hat{\boldsymbol{C}}$ are positive, which means that only the first quadrant of Cartesian space needs to be taken into consideration. As a measure of similarity the moments of the spectral (chromatic) vector distances are used.

$$C_4 = 1 - \frac{1}{MN}\sum_{j=1}^{M}\sum_{i=1}^{N}\left\{\frac{2}{\pi}\cos^{-1}\left(\frac{\langle\boldsymbol{C}(i,j),\hat{\boldsymbol{C}}(i,j)\rangle}{\|\boldsymbol{C}(i,j)\|\|\hat{\boldsymbol{C}}(i,j)\|}\right)\right\}, \tag{6.11}$$

where the normalization factor $2/\pi$ is related to the fact that the maximum difference attained will be $\pi/2$ and $\langle\boldsymbol{C}(i,j),\hat{\boldsymbol{C}}(i,j)\rangle$ is the inner product given by

$$\langle\boldsymbol{C}(i,j),\hat{\boldsymbol{C}}(i,j)\rangle = R(i,j)\hat{R}(i,j) + G(i,j)\hat{G}(i,j) + B(i,j)\hat{B}(i,j), \tag{6.12}$$

where $R(i,j)$ and $\hat{R}(i,j)$ represent the red component of the pixels at position (i,j) in $\boldsymbol{C}$ and $\hat{\boldsymbol{C}}$ respectively. The same holds for the green and blue components. The denominator in 6.11, expresses the product of the Euclidean norm

of $\boldsymbol{C}$ and $\hat{\boldsymbol{C}}$ respectively, which is given by

$$\|\boldsymbol{C}(i,j)\| = \sqrt{R(i,j)^2 + G(i,j)^2 + B(i,j)^2}. \tag{6.13}$$

6.2.4 Using Metrics in the Image Analyzer

The above metrics all try to estimate the degree of similarity between two images in a relatively simple manner. Therefore, these metrics have been implemented in the image analyzer. Additionally, the above metrics are easy to extend in order to account for images from multiple viewpoints. The image collector, which was described in detail in the previous chapter, captures a collection of images from a number of different camera positions. This gives multiple images of the same scene, and the above image metrics can be applied to the entire set of images. These per image differences can be combined into a single measure. For all of the image metrics considered in this project, it is natural to sum the differences over all pixels in all images. Formally, given two sets of L images $S = \{s_k\}$ and $S' = \{s'_k\}$ of dimension $M \times N$ pixels, the absolute difference between two sets of images is computes as

$$\Delta D_1(S, S') = \frac{1}{LMN} \sum_{k=0}^{L-1} \sum_{j=0}^{M-1} \sum_{i=0}^{N-1} (|s_{ijk} - s'_{ijk}|). \tag{6.14}$$

The same procedure applies to the other metrics mentioned above.

In general, the metrics described above all have their origin in digital image compression, where a compressed image is compared to its non-compressed counterpart. We use the metrics in a different context, which has more focus on the way the visual difference is perceived by a person observing the virtual scene environment contained in a game or a simulator. Unfortunately, we have not been able to find research material on how these metrics incorporate the human visual effect in such scenarios, and such an investigation has been considered outside the scope of this project.

6.3 Visual Perception Of Images

The tool we have developed in this thesis relies on human assistance regarding the evaluation of the collected images. Although, the image analyzer tries to estimate the level of similarity between collected scene images, these estimates are all based on statistical pixel differences rather than overall perceptual deviation between the images. However, incorporating the way humans perceive visual differences in images into the image analyzer, is not a trivial task. Image attributes like, contrast, acuity, object border, and color are trivial to perceive for the human eye, but capturing and comparing these in a virtual scene environment is complicated.

The human visual system is an extraordinary imaging device, capable of processing light within broad ranges of spectral frequency, spatial resolution, and light intensity levels. These capabilities far exceed the range of outputs from conventional computer monitors and paper prints - the two most commonly used media in computer graphics. However, even within these narrow ranges, the human eye does not perceive individual pixels as linear functions of frame buffer

intensities, nor do the eyes treat an image as a height field of intensity values. Rather, several complex mechanisms throughout the visual cortex transform incoming light into signals that are interpreted by the brain in a manner much different from the physical light that strikes the retina.

In order to implement an image analyzer component, which takes into account some of the effects perceived by the human eye, different image attributes must be considered. The most significant of these are described in the following.

6.3.1 Contrast

Contrast is the local change in brightness and is defined as the ratio between average brightness of an object and the background brightness. The human eye is logarithmically sensitive to brightness, implying that for the same perception, higher brightness requires higher contrast.

Apparent brightness depends very much on the brightness of the local background; this effect is called conditional contrast. Figure 6.2 illustrates this with two small squares of the same brightness on a dark and light background. Humans perceive the brightness of the small squares differently.

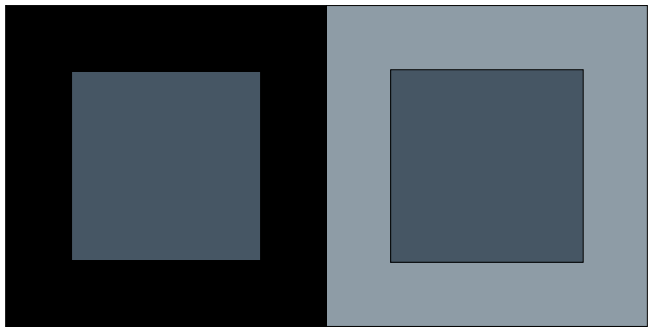
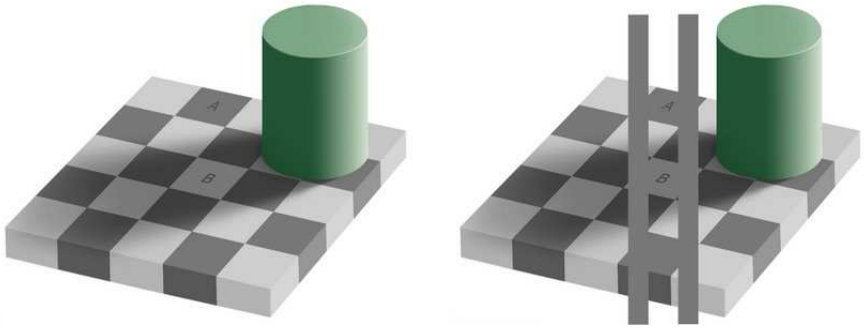


Figure 6.2: Conditional contrast - illusion of brightness.

To represent the sense of three dimensions or scene depth in computer graphical virtual environments, shading of objects are used to create a sense of perspective. Using this approach can create an illusion of how contrast and brightness are perceived. This is illustrated in figure 6.3 created by Edward H. Adelson [8], where a cylinder is placed on a board containing dark and light squares. The lightning condition in the scene creates a shadow effect across the board, which makes square A looks dark and square B looks light, but this an illusion - in fact the two squares have the same color, which can be seen from the two vertical gray bars.

The image metrics discussed above do not take any special precautions regarding contrast and brightness levels in the images. This weakness naturally affects the image analyzer, since these parameters are used in most scene representations found in virtual environments. However, incorporation contrast and brightness into the image metrics has been considered outside the scope of this thesis.



(a) (b)

Figure 6.3: Shading and brightness affect how a scene is perceived by the human eye. (a) Shadow across a board makes the color of the squares look different. (b) The gray bars show that the two squares have the same color.

6.3.2 Acuity

Acuity is the ability to detect details in the image. The human eye is less sensitive to slow and fast changes in brightness in the image plane but is more sensitive to intermediate changes. Acuity also decreases with increasing distance from the optical axis.

Resolution in an image is firmly bounded by the resolution ability of the human eye; there is no sense in representing visual information with higher resolution than that of the viewer. Resolution in optics is defined as the inverse value of the maximum viewing angle between the viewer and two proximate points which humans cannot distinguish, and so fuse together.

6.3.3 Object Border

Object borders carry a significant amount of information. Boundaries of objects and simple patterns such a blobs or lines enable adaptation effects similar to conditional contrast, mentioned above. The Ebbinghaus illusion is a example of the effects, which is shown in figure 6.4. Here two circles of the same diameter are placed on the same horizontal axis. The first circle is surrounded by four circles with a diameter smaller than itself, and the second circle is surrounded by four circles with a larger diameter than itself. This arrangement creates the illusion that the two center circles have a different diameter.

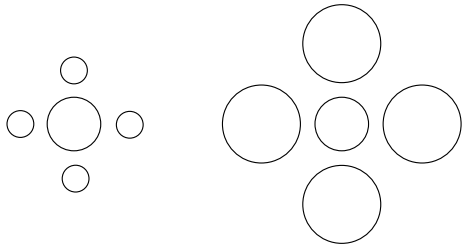


Figure 6.4: The Ebbinghaus illusion.

The image analyzer does not included special algorithms to check if the scene images under investigation have similar borders. However, deviation in object borders will be expressed in the similarity computations performed on the scene images.

An intuitive approach to determine if two scene images contain similar objects and these have similar borders, is to divide the images into parts that have a strong correlation with objects or areas of the virtual world contained in the images. This approach corresponds to image segmentation, which results in a set of disjoint regions corresponding uniquely with objects in the scene image. Comparing the scene images, with respect to object borders, could then be done by comparing each discrete region to the corresponding reference region.

6.3.4 Color

Color is very important for perception, since under normal illumination conditions the human eye is more sensitive to color than to brightness. Historically, image processing and computer vision have mostly been concentrated around processing monochromatic images. The reason for this has been the additional cost of suitable hardware for processing color images or multi-spectral images.

Color is connected with the ability of objects to reflect electromagnetic waves of different wavelengths. The chromatic spectrum spans the electromagnetic spectrum from approximately 400 nm to 700 nm. The human eye detect colors as combinations of the primary colors red, green, and blue, which have for the purpose of standardization have been defined as 700 nm, 546.1 nm, and 435.8 nm respectively.

Display hardware will generally deliver or display color via an RGB model (referring to red, green, and blue). This means that a particular pixel may have associated with it a three-dimensional vector (r, g, b) which provides the respective color intensities, where $(0, 0, 0)$ is black, (k, k, k) is white, $(k, 0, 0)$ is pure red, and so on. The RGB model is visualized in figure 6.5.

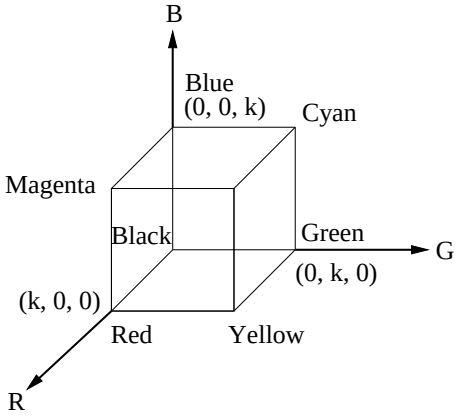


Figure 6.5: RGB color space.

The scene images captured by the image collector are all color images, which makes it possible to process them using color image algorithms. Color image algorithms are in general more advanced than greyscale image algorithms, since

in greyscale images each pixel carries only intensity information which can be represented using 8 bits. This means that greyscale images often are used in image processing, where human visual perception is insignificant. However, the strategy in this thesis relies on human intervention in order to determine if scenes are identical, therefore using greyscale images could make it possible for a programmer to make false decisions. Suppose the collected scene images were greyscale images showing a character in a virtual environment. If the texture of the character was changed in between two runs of the image collector and image matching was applied afterwards, a greyscale comparison would not catch the differences and the programmer looking at the collected scene images would not be able to observe the differences. Therefore, it was decided not to perform this conversion, since the possibility of detecting variation in colored scene objects would be lost.

The color images captured by the image collector are stored as 24 bit images with one byte for each channel. It was discussed whether converting the color images to greyscale images could be used in some situations and thereby making the image matching algorithms more efficient, since less processing would be required. Using greyscale images to check if scene objects were positioned correctly and color images to check for texture differences, would be a way to make the scene image analyzer more efficient.

The images processed by the image analyzer are represented in the RGB color space. To archive matching estimates which lie closer to human visual perception a different color space could have been used. A color space especially suitable to capture visual perception is the CIE XYZ 1931 color space or simply the CIE color space [Jain, 1989]. A conversion from the RGB color space to the CIE color space can be performed by the following equation:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \begin{bmatrix} 0.490 & 0.310 & 0.200 \\ 0.177 & 0.813 & 0.011 \\ 0.000 & 0.010 & 0.990 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}, \tag{6.15}$$

Other color spaces, which imitate visual perception better than the RGB color space, include HSV (hue, saturation, value) and the CIE LUV color space. However, it was decided to use the RGB color space due to its simplicity.

6.3.5 Perceptual Motivated Metrics

The image analyzer implemented in this thesis relies upon the equations based on pixel difference measures and correlation based measures. These metrics have been chosen due to their relatively simple nature and because they to a large extend could be implemented in the image processing framework used in the project. However, these metrics are all based on statistical analysis rather than visual perception. Image metrics that tries to resemble human visual perception do exist, and the ideas behind them are explained in the following.

During the last century, research in psychology and physiology has lead to remarkable advances in understanding the workings of the human visual system. Drawing upon this information, research in computer vision have developed mathematical models that predict how humans perceive simple patterns of light, which can sometimes be generalized to more complex visual scenes. Recognizing that such models are extremely powerful tools for comparing digital images, several perceptually motivated image metrics have emerged in computer vision

and graphics fields. Currently, two of the most perceptually accurate metrics for comparing images are the Visual Difference Predictor and what is know as the Sarnoff model [Lindstrom, 2000]. These two metrics are built on the notion of "just noticeable differences", that is differences between two images that are just above the threshold of detectability. Both of these metrics attempt to emulate some of the stages in the visual system, such as spatial frequency and orientation decomposition, correction of variable sensitivity to different light and contrast levels, suppression of patterns due to visual masking, and so on. These models are motived by the assumption that applying such transforms to images will result in a representation that is closer to the one used in the higher level processing stages of the human visual system, and that measuring differences between these representations will more accurately reflect how humans actually perceive images differences.

The metrics discussed above provides a fair indicator of the similarity for many geometric models. Nevertheless, there are models, and images in general, for which these metrics do not reflect well how differences between two images are perceived. The reason for this difference in correlation is that the human visual system does not process images as though they were collections of independent pairs of linear pixel intensities, which the above metrics does. Rather, the images that strike the retinas undergo some fairly complex processing before they are interpreted by higher level systems in the visual cortex. These processes can lead to suppression of some image differences and magnification of others.

In [Ramasubramanian et al., 1999] a procedure for creating a physical error metric, that predicts the perceptual threshold for detecting artifacts in scene features, is presented. Built into this metric is a computational model of the human visual system's loss of sensitivity at high background illumination levels, high spatial frequencies, and high contrast levels.

Incorporating metrics, which exploit how humans perceive images, into the image analyzer would probably make it even more reliable. However, the pixel based metrics and the correlation based metrics provide a fair indicator of the degree of similarity. In this project the primary focus has been to investigate the elements needed in order to create a tool for testing graphical virtual environments, and implement the proposed strategies to achieve proof of concept. Therefore, more advanced algorithms, which take into account the issues in visual perception and human interpretation of images, have not been the primary focus in this thesis. However, such algorithms could easily be implemented without affecting the overall design of the tool.

6.4 The Image Metric Analyzer Component

This section describes the design and implementation of the image metric analyzer responsible for analyzing the collected images taken from multiple scene views using image metrics to measure the degree of similarity between the scenes.

6.4.1 Design

The image collector component is only responsible for collecting multiple scene views in virtual Delta3D environments, and does not have any mechanisms

to determine whether the collected scene images match previously collected images. The task of comparing the collected images, using the image metrics described earlier, is delegated to the a separate component, which in this project is a human tester. The image processing and metric calculations could have been incorporated into the image collector module, but this approach was not investigated further since keeping these modules independent was consider more flexible.

The image metric analyzer is responsible for making image comparison using the previously described image metrics. The image metric analyzer is a self contained module in the sense that is does not have any direct coupling to Delta3D. Instead, it relies on the functionalities included in the Image Processing Library 98 (IPL98). The IPL98 library was analyzed in detail during the preliminary work conducted before writing this thesis, and details about the features provided by IPL98 are included in [Horn and Grønbæk, 2008]. IPL98 provides C++ image container classes and basic image processing algorithms. Unfortunately, the library does not contain image metrics algorithms, so these have been implemented to provide the necessary features. As mentioned above, IPL98 provides classes to represent BMP images, the only class used by the image analyzer is the class `ipl98::CImage`. This class makes it possible to read and write BMP images from and to disc respectively. Additionally, the class has methods to extract pixel informations like RGB channel colors and pixel values from BMP images.

6.4.2 Implementation

The image metric analyzer is implemented in a single class, which contains methods necessary to implement the metrics discussed in the sections 6.2.2 and 6.2.3. There are only minor differences between the implementation of one metric compared to another. All metrics are given two references, of type `ipl98::CImage`, to standard IPL98 bitmap images. These references correspond to the images being compared.

Listing 6.1 shows the implementation of the cross correlation metric. Here each pixel in the two images is visited and the RGB values are extracted and used in the similarity calculation corresponding to equation 6.9.

6.4.3 Test

The following presents some test results, showing how the image metric analyzer estimates the degree of similarity between selected scene images from the WalkingSoldier application.

Test 1

Figure 6.6 shows two images captured from the WalkingSoldier application. Each image is captured in a separate run of the application. Both images are captured at the same in-game position and with the same camera transformation in the virtual environment. Additionally, the simulation time is identical in both runs, which should make the two images identical. By visually inspecting each of the images, it becomes clear that they are identical which also should be evident from the metric calculations. Table 6.1 presents the results obtained

```
double ImageMetricAnalyzer::computeCrossCorrelationImageMetric(const
ipl::CImage &img1, const ipl::CImage &img2)
{
  unsigned int i;
  unsigned int j;
  const unsigned int n = img1.GetWidth();
  const unsigned int m = img1.GetHeight();
  double redConNum = 0.0;
  double greenConNum = 0.0;
  double blueConNum = 0.0;
  double redConDen = 0.0;
  double greenConDen = 0.0;
  double blueConDen = 0.0;
  for (i = 0; i < n; i++)
  {
    for (j = 0; j < m; j++)
    {
      redConNum += img1.GetRedComponent(i, j) * img2.GetRedComponent(i, j);
      greenConNum += img1.GetGreenComponent(i, j) * img2.GetGreenComponent(i, j);
      blueConNum += img1.GetBlueComponent(i, j) * img2.GetBlueComponent(i, j);
      redConDen += pow((double)img1.GetRedComponent(i, j), 2);
      greenConDen += pow((double)img1.GetGreenComponent(i, j), 2);
      blueConDen += pow((double)img1.GetBlueComponent(i, j), 2);
    }
  }
  return ((redConNum / redConDen + greenConNum /
  greenConDen + blueConNum / blueConDen) / 3.0f);
}
```

Listing 6.1: Method `computeCrossCorrelationImageMetric` computes a similarity estimate between two images of type `ipl98::CImage`.

(a) Soldier from first run. (b) Soldier from second run.

Figure 6.6: Two identical scene images captured in two separate runs.

by applying the image metric analyzer to the images. From the values of the metric calculations, it is evident that the two images are similar. The mean square metric and the root mean square metric both gives a 0.0 value indicating a 100 percent similarity. The other metrics all give a value of 1.0, which also indicate 100 percent similarity.

Test 2

Figure 6.7 shows two images captured from the WalkingSoldier application. Each image is captured in a separate run of the application. In the first run, the soldier carries a weapon and in the second he does not. Both images are captured from the same position in-game and with the same camera transformation in the virtual environment. By inspecting the images, it becomes clear that they are not identical due to the missing weapon, which should be evident from the metric calculations. Table 6.2 presents the results obtained by applying the image metric analyzer to the images shown above. Both the mean square metric

Metric	Value
Mean square	0.000
Root mean square	0.000
Structural content	1.000
Cross correlation	1.000
Angle moment	1.000
Czenakowski	1.000

Table 6.1: Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes, which are completely identical.

(a) Soldier with weapon. (b) Soldier without weapon.

Figure 6.7: Two almost identical scene images captured in two separate runs. However, the soldier carries a weapon in the first image, which he does not in the second image.

and the root mean square metric, gives the same result, namely 30.232. However, the other metrics all give similarity estimates close to 1.0, which indicates that the scene images are very similar - without being identical.

Test 3

Figure 6.8 shows two images captured from the WalkingSoldier application. Each image is captured in a separate run of the application. In the first run, the soldier carries a helmet and in the second he does not. Both images are captured from the same position and with the same camera transformation in the virtual environment. By inspecting the images, it becomes clear that they are not identical due to the missing helmet , which should be evident from the metric calculations. Table 6.3 presents the results obtained by applying the image metric analyzer to the images shown above. Again, both the mean square metric and the root mean square metric, give the same result, namely 19.251 and the other metrics all give similarity estimates close to 1.0, which indicates that the scene images are very similar - without being identical.

Test 4

Figure 6.9 shows two images captured from the WalkingSoldier application. Each image corresponds to a separate run of the application. In the first run,

Metric	Value
Mean square	30.232
Root mean square	30.232
Structural content	0.998
Cross correlation	0.996
Angle moment	0.985
Czenakowski	0.980

Table 6.2: Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes, where the soldier carries a weapon and where he does not.

(a) Soldier with helmet. (b) Soldier without helmet.

Figure 6.8: Two almost identical scene images captured in two separate runs. However, the soldier carries a helmet in the first image, which he does not in the second image.

the soldier is present and fully equipped, carrying helmet and weapon. In the second run the soldier is not present at all. Both images are captured from the same position and with the same camera rotation in the virtual environment. Table 6.4 presents the results obtained by applying the image metric analyzer to the images shown above. Again, both the mean square metric and the root mean square metric, give the same result, namely 220.980. This is a relative high value compared to the ofter tests. The reason is that the soldier constitutes a larger pixel area in the scene compared to the area of the helmet or the area of the weapon. However, this observation does not entirely correspond with the other metrics, which give slightly different results - ranging from 0.915 to 0.973.

Test 5

Figure 6.10 shows two images captured from the WalkingSoldier application. Each image is captured in a separate run of the application. By inspecting the images, it becomes clear that the position and orientation of the soldier are different in the two images. Table 6.5 presents the results obtained by applying the image metric analyzer to the images shown above. Both the mean square metric and the root mean square metric give the same result, namely 58.02. This result is difficult to interpret, since it can not directly be transferred to

Metric	Value
Mean square	19.251
Root mean square	19.251
Structural content	0.971
Cross correlation	0.996
Angle moment	0.988
Czenakowski	0.986

Table 6.3: Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes where the soldier carries a helmet and where he does not.

(a) Soldier from first run. (b) Soldier from second run.

Figure 6.9: Two scene images captured in two separate runs. However, the soldier is present in the first image, which he is not in the sceond image.

a percentage similarity estimate. However, the other metrics all give similarity estimates between 0 and 1, which indicate total dissimilarity or total similarity respectively. As shown, these estimates span from around 0.97 to 0.99 which indicates that the images are almost identical. A programmer or a tester might find it difficult to judge whether the result of such a metric calculation should result in the respective regression test being considered a pass or fail. Fortunately, the image metric analyzer has functionality to visualize the difference between two scene images. Figure 6.11 shows the difference between the two images in figure 6.10. As shown, most of the background is black which indicates total similarity between the pixels in the respective images. The pixel area around the soldier overlaps in the difference image, which means that the position and rotation of the soldier only partially coincide in the two images. The image was created by subtracting the two images. The method `computeImageDifference` contained in the image metric analyzer class is responsible of subtracting the collected scene images.

6.4.4 Conclusion

The conclusion after having tested the image analyzer, is that programmers and testers might find it difficult to judge whether a test should pass or fail based on the metric calculations alone. The above tests only represent a small subset of the situations the regression test tool could be used in. Therefore, general

Metric	Value
Mean square	220.980
Root mean square	220.980
Structural content	0.973
Cross correlation	0.961
Angle moment	0.923
Czenakowski	0.915

Table 6.4: Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes where the soldier is present and where he does not.

(a) Soldier from first run. (b) Soldier from second run.

Figure 6.10: Two scene images captured in two separate runs. However, the soldier is oriented differently in the two images.

statements about the interpretations of the metric values can not be made at this point. More testing is required to fully evaluate the image analyzer. However, the above tests have shown some situations, in which the image analyzer catches scene image differences and visualizes these differences so a programmer or tester can act accordingly.

6.5 Test Reporter Tool

The following sections describes the Test Reporter tool, and show how the tool presents test results created using the VETS tool, the Image Collector component, and the Image Analyzer component. A couple of examples of generated test reports are shown, and their relevance evaluated. To explain the use of the reporter tool in entirety, the following section first explains the use of the test case selection and execution process, since the reporter tool is invoked as a part of the test execution tool.

6.5.1 Test Case Selection and Execution

By using NAnt, to execute test cases, it has been possible to create a flexible configuration that can be used when executing the application under test. The VETS tool, which is also responsible for test input and output generation, is designed to accept the necessary external input, in the form of runtime argu-

Metric	Value
Mean square	58.02
Root mean square	58.02
Structural content	0.997
Cross correlation	0.995
Angle moment	0.981
Czenakowski	0.978

Table 6.5: Comparing metric computations in the WalkingSoldier application. The numbers express the difference between two scenes where the soldier is oriented differntly.

Figure 6.11: The difference in position and rotation is calculated by the image analyzer and presented in a "difference image".

ments. This sets the internal configurations of the component on what test case that should be run, and on what to do with test output. Therefore, the use of NAnt allows this project to create execution scripts specifying a list of tests that must be run for each application. After each test case, the output can be processed.

An important consideration to take when creating an automated testing system based on a capturer-and-replay tool, is that it is likely that the execution of a test case will be done in real time. While the Delta3D framework, and in particular the Game Manager framework, implement functionality to increase the time scale of the simulation, it is not something that has been considered in depth in this project. This means that if a tester performs a test capture that last 10 minutes in real time, the execution of the test case will also last 10 minutes. In addition, the construction used in this project, where screen shots are captured, assumes that the application that is being executed is displayed on a monitor. This is due to the fact that the underlying Delta3D camera models are assuming that a monitor is always present. This means that it is not possible to run application tests on a headless server, but that the tests must be executed on a real machine with a graphics processor and attached to a monitor. When a test is executed, Delta3D also expects the monitor to be on, the application window to be focused, and that no other windows on the desktop are placed in front of the application window. All these assumption can be summarized to: the tests must be run one at a time, in real time, on a developer machine, which must have its monitor turned on. These assumption quickly become encumbering in the daily testing process, especially if many tests

```
44    <!--
45          Select test cases to be executed by adding properties and
46          target calls.
47      -->
48    <target name="nightly_test" depends="nightly_test_prepare">
49      <!-- TEST 1 - general keyframe test case -->
50      <property name="test_name" value="test_1" />
51      <call target="run_test" />
52
53      <!-- DO NOT RUN THIS TEST
54      <property name="test_name" value="test_2" />
55      <call target="run_test" />
56      -->
57
58      <!-- TEST 3 - test full screen tank movement -->
59      <property name="test_name" value="test_3_fullscreen" />
60      <call target="run_test" />
61    </target>
```

Listing 6.2: Test case selection

are to be executed. To relieve this situation, the Cruise Control .NET scheduled build is used to control when tests are executed. The Cruise Control .NET server is configured to run tasks at pre-configured intervals, and when specific conditions are found to be true. By configuring the Cruise Control .NET to execute the constructed NAnt execution scripts in a nightly build routine, an automated regression testing procedure was constructed for the test components developed. This was done using NAnt execution scripts.

Selecting Test Cases

The organization of test cases in a folder structure was shown in section 3.4.1. This structure allows a very simple NAnt scripting processes to be made, where the relevant test cases can be listed. Listing 6.2 shows an excerpt of the complete NAnt script for testing the GMTank application. On line 50, 54 and 59 the property `test_name` is set, and each time followed by a call to the target `run_test`, which is discussed in more detail in the next section. The `run_test` method uses the `test_name` property to locate a directory with test information, and then executes that test. Thus, by placing test cases in sub-folders with specific names, it is possible to use the NAnt script to only execute selected test cases. In listing 6.2 the test case named `test_2` on line 54 is not included in the test case executions, since it is uncommented by the line above.

Executing Test Cases

Listing 6.3 shows the `run_test` target, which is responsible for executing a test case. The target only executes one test case at a time, and must be invoked once for each case that should be executed. On line 91 the application is executed using the Delta3D tool *gamestartd.exe*, which is a build-in tool for executing applications that implement the GM framework. The VETS tool, that is build into the application, receives and parses the arguments specified on line 93, which includes the location to find the test input in, and a name to store the test output under. On line 91 the working directory is set to the location of the applications executable, since this is where the oracle information will be output to. After the execution of the application, and moving of the results, the Report Generator tool is invoked in line 106. The responsibility of the Test Reporter is explained in the following section.

```
71    <!--
72     Execute the actual tests by running the executable
73     The "test_name" property must be set to identify the test to execute
74    -->
75    <target name="run_test">
76
77      <!-- Set local variables used in the test-->
78
79      <!-- the current build label (from CCNET) -->
80      <property name="build_label" value="${CCNetLabel}" />
81      <!-- where to store the results -->
82      <property name="dir.output" value="${project.dir.results}/${test_name}/${
            build_label}" />
83      <!-- the name of the log file to save -->
84      <property name="log.output" value="${test_name}_log.txt" />
85
86      <!-- Create a folder to store the test results in -->
87      <!-- <delete dir="${dir.output}" if="${directory::exists(dir.output)" /> -->
88      <mkdir dir="${dir.output}" />
89
90      <!-- Execute the program by running the gamestart(d).exe application -->
91      <exec program="gamestartd" workingdir="${project.dir.build}">
92        <!-- Arguments for gamestart - 1:the_library 2:the_file_to_log_to 3
              :the_file_to_replay -->
93        <arg line="TutorialGameActors_${log.output}_../${project.dir.tests}/${
              test_name}/${test_name}.txt" />
94      </exec>
95
96      <!-- Move the test results to the storage folder -->
97      <move todir="${dir.output}" overwrite="true">
98        <fileset basedir="${project.dir.build}">
99          <include name="*.jpg" />
100         <include name="${log.output}" />
101         <include name="delta3d_log.html" />
102       </fileset>
103     </move>
104
105     <!-- Invoke the Report Generator -->
106     <call target="create_html" />
107
108   </target>
109
110   <!--
111       The Report Generator tool.
112       This call the tool, which is a Java executable, with input
113       arguments:
114       1 - name of the output report (same as test name)
115       2 - source folder of the test case
116       3 - destination folder of the output files
117       Notice that the Report Generator automatically invokes the Image
118       Analyzer component internally.
119     -->
120   <target name="create_html">
121
122     <!-- Set local variables used in the test-->
123     <!-- the tool is checked out and compile here by CCNET -->
124     <property name="file.generator" value="../ReportGenerator/dist/HTMTLImgComparer.
            jar" />
125
126     <!-- Generate a html file to view the results -->
127     <exec program="java">
128       <arg line="-jar_${file.generator}_${dir.output}/${test_name}.html_${project.
              dir.tests}/${test_name}_${dir.output}" />
129     </exec>
130   </target>
131
132 </project>
```

Listing 6.3: Test case execution

6.5.2 Test Reporter

The Test Reporter tool implements functionality to assist a human oracle in performing the comparison and identification steps contained in the test procedure used in this project. This is done by running the Image Analyzer component, which was described earlier, and subsequently generate a HTML page, which presents the referenced images, the oracle information, the subtracted images, and finally the image comparison metrics in a simple side-by side structure, allowing a tester to quickly get an overview of the results.

Figure 6.12 shows an execution of the WalkingSoldier application, which has been executed with a recorded test input using the test case execution script. The first column shows the referenced images collected during the test

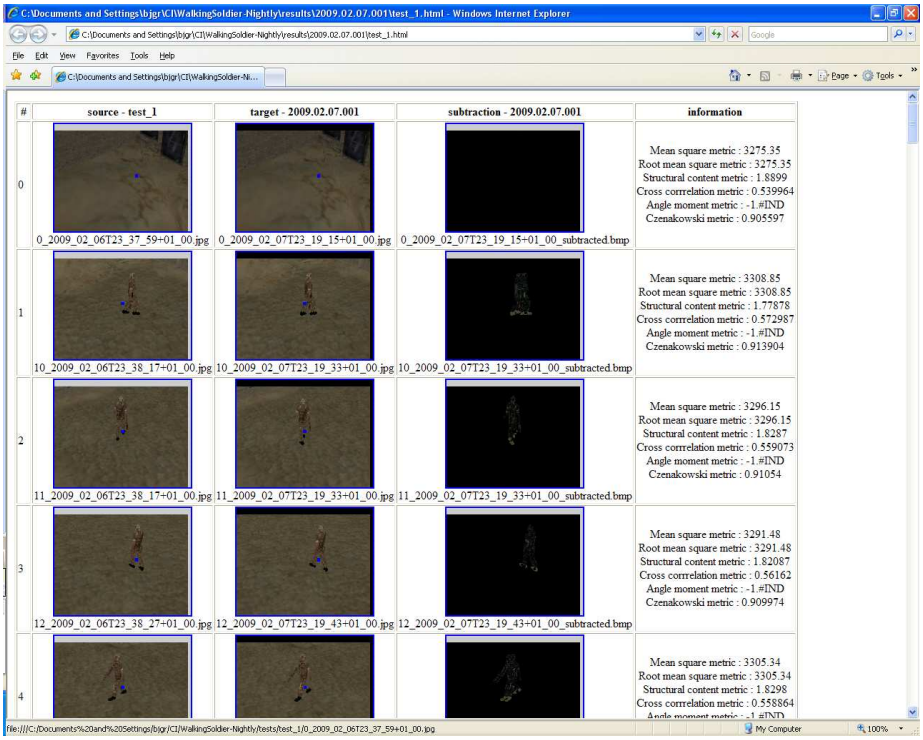


Figure 6.12: Report Generator and the WalkingSoldier test

input recording. The second column shows the associated oracle information image captured during the execution of the test input. Column three shows the two images subtracted, resulting in black coloring where the images are identical. The final column presents the image metrics. The subtracted images in this test indicate, that the two executions are very similar, except for the area just surrounding the solider. This is as expected, since the soldier is rendered differently on every execution. If a human tester is clear that this is the case, he can easily confirm that the other parts of the images are identical.

Figure 6.13 shows an execution of the GMTank application. Similar to the WalkingSoldier test, a test report is generated. In this report it is clear from the subtracted images that they are not completely identical, but almost similar.

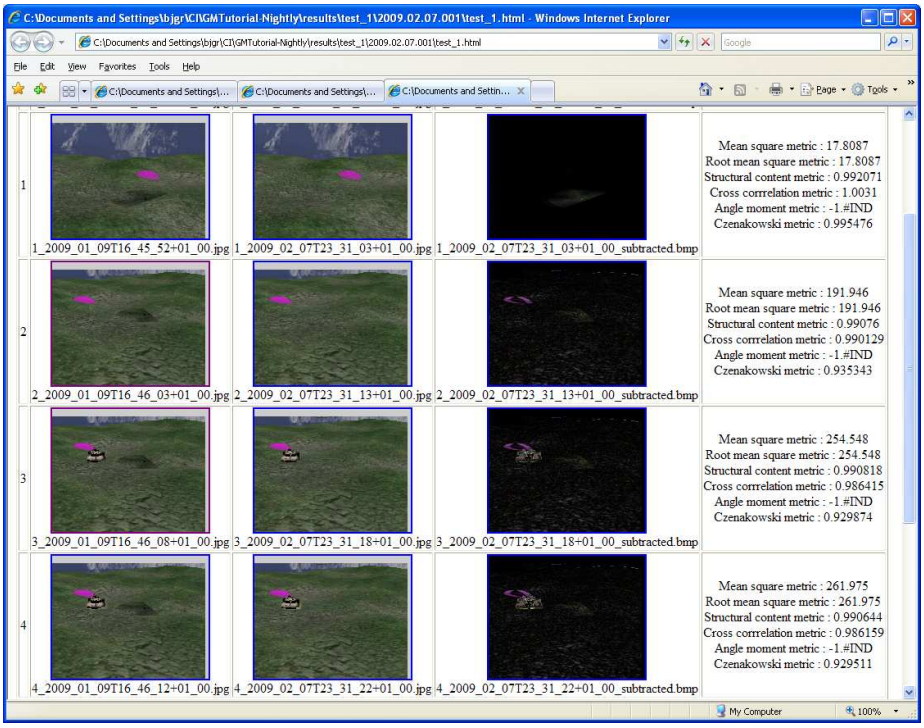


Figure 6.13: Report Generator and the GMTank test

The tank in particular is differently placed on the reference and oracle images. This is due to the timing effect, as described in section 4.4.2, where in this test, the original reference images are used, rather than a timing corrected version. Again, the subtracted images can be used to spot areas of particular difference faster than just the reference and oracle images alone.

Figure 6.14 shows a report generated by a test case execution in the GMTank application. This is a different test case than the one shown in the report in figure 6.13, where there is a somewhat subtle error introduced in the test input. The result of the error is that in the third set of images the landscape of the oracle information is rotated 90 degrees, relative to the reference image. This can be spotted by the sphere in the top center of the original missing in the oracle information, but careful inspection also reveals that the corners of the terrain are not placed identically on the images. In a quick inspection these two discrepancies might not be identified, but the subtracted image clearly indicates that there is a problem with the landscape, especially around the top left corner. In addition, the presence of the colored sphere in the subtracted image indicates a problem around this area.

6.5.3 Conclusion

The Report Generator tool was implemented with the intention of showing the basic functionality of the tools developed in this project, and with the intention to assist a human oracle procedure. While the current implementation is almost as simple as possible to envision, the Report Generator succeeds in illustrating

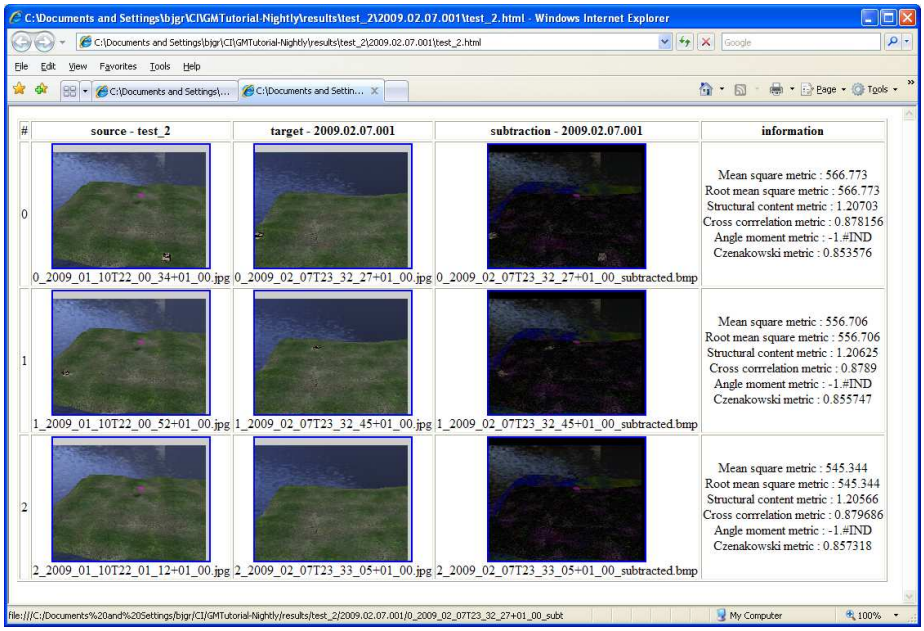


Figure 6.14: Report Generator and the GMTank test 2

the decrease in work that a tester must perform. In addition, the report generated makes use of all test tool elements constructed for this project, both the VETS tool, the Image Collector component and the Image Analyzer component.

By using the NAnt scripting system for selecting test cases, executing tests, and generating reports, it has been shown that the tools developed in this project can be integrated into existing industrial software testing tools.

Chapter 7

Perspectives

This chapter concludes the work done in this thesis. First, a section mentioning important existing research related to regression testing in graphical virtual environments is presented. Secondly, the areas which were researched and covered in this thesis, are reviewed in the context of further work and research. Finally, an elaboration of the testing procedure found and the developed framework extensions are presented.

7.1 Related Work

The tool created in this thesis is composed of a variety of different modules, each concerned with a different area of testing and with different responsibilities. Information about studies, which use a similar approach to graphical regression testing, as the approach developed in this project, has been difficult to find. Literature in most areas in isolation can of course be found.

- The work on object oriented frameworks and the issues involved in extending existing frameworks, described in [Van Gurp and Bosch, 2000] and [Bosch et al.,], have been very useful, and informative on how to expand existing implementations of frameworks.
- The material about using multiple scene views in 3D virtual environments, was greatly inspired by the work of Peter Lindstrom, who has used multiple scene views to perform image rendering of objects in virtual environments. This work is described in [Lindstrom et al., 1996],[Lindstrom, 2000], and [Lindstrom and Turk, 2000].
- The image analyzer uses image metrics to compare the collected scene images with stored references. Most of the metrics used in the implementation are not specially suited for comparing scene images captured in virtual environments. Instead, these metrics are based on statistical correlation between images. A comprehensive survey of statistical image metrics can be found in [Avcıbaş et al., 2002]. Additionally, authors have investigated the metrics used in this implementation in the context of image distortion and image compression, where the metrics have been used to compare distorted images with their undistorted counterparts, or comparing compressed images with uncompressed images. Details about this work can

be found in [Ives et al., 1999] and [Sun and Li, 2005]. Work about image metrics that incorporate effects like brightness, color, contour, etc. into the similarity estimates, are described in [Toet and Lucassen, 2003]. Advanced image metric algorithms, which considers human perceptual effects are described in [Ramasubramanian et al., 1999].

- GUI testing and test oracle design is described in much detail and in many different articles, especially by [Xie and Memon, 2007], [Memon and Soffa, 2003] and [Memon, 2002].
- Specific testing of virtual reality interfaces in both simulations and game development has turned out to be a subject which not many authors have written about. A few sources which has been relied on for inspiration is this project is [Bierbaum et al., 2003] and Noel Llopis with [Llopis, 2003] and [9].

7.1.1 ANN's as Automated Oracles

[Vanmali et al., 2002] suggests creating automated oracles using Artificial Neural Networks (ANN). ANN's have already been used in several software testing fields, for example for evaluating the effectiveness of generating test cases for exposing software faults. By creating a multi layered ANN and training the network on an existing software, to create a similar output from a set of input data, as the original software with the same input. By comparing the output of a modified version of the software and the output of the ANN it is possible to detect faulty behavior in the software. New versions of the software is regression tested in this way. A comparison tools treats the output from the modified software and from the ANN and makes a decision on the correctness of the software output.

[Vanmali et al., 2002] summarizes several reasons to prefer an ANN model of the software, rather than the original software it self, when generating output data:

1. The original software may become unusable due to platform and hardware changes. This could also occur as the result of changes in libraries or other third-party applications.
2. It is possible to train an ANN with only the input that are relevant to a given test, rather than creating a full set of input data for the original software to process, thus saving computational resources.
3. It may not be possible to save an exhaustive set of test cases and outputs from the original software in a practical manner.
4. Using an ANN may provide an additional output indicating the reliability of the output. This is valuable in situation where the output of the original and the modified both would be faulty, and a fault could remain undetected.

Figure 7.1 shows an overview of the evaluation phase using the modified software and the ANN. Both are presented with the same set of input data and each produce a set of output data which are subsequently compared to evaluated the correctness of the software output.

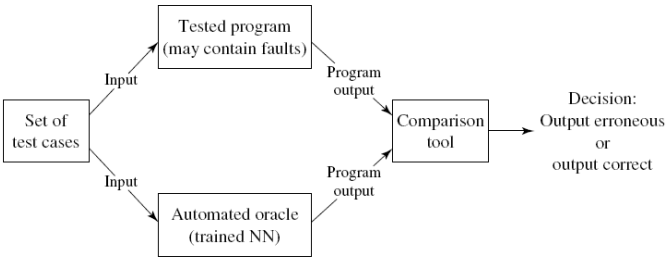


Figure 7.1: ANN evaluation phase

7.2 Future Work

Many areas covered in this thesis could be explored further both in detail and in added functionality. While the time frame available for this project has not allowed all topics to be treated in entirety, the follow section presents a number of key areas that could be elaborated on, if more work were to be done in the focus areas of this project.

Image Analysis Incorporating more advanced image matching methods into the image analyzer, would reduce the manual work performed in the oracle procedure, and thereby make the tool less dependent on human judgement. Especially, more advanced image metric algorithms which take into account the human visual system, are necessary in order for the image analyzer to rely more on how images are perceived than statistical pixel differences.

The image analyzer relies entirely on the precision of the metric calculations. By adding different image matching algorithms to the image analyzer, a more versatile matching cold be made. Thereby, elimination some of the uncertainty found when the tester should judge whether a test should pass or fail.

Test Support Tool The test support tool, VETS, developed in this project is only a minimal implementation, providing testers with the smallest amount of test input, to present to later test executions. Apart from fixing the timing bug described previously, further work on this tool will obviously add more default functionality to the capture-and-replay logging part, providing a more feature rich blackbox component for testers to utilize.

Another area to investigate and develop further would be the state capturer functionality described in section 2.3, and shown in figure 2.5. It would be very interesting to implement and test if such an execution and state testing strategy could be combined with the image capturer and comparison approach used in this project.

Further exploration of the build AAR tool as a replacement or supplement to the VETS tool is also an area of interest. Even if the tool is specific for Delta3D, the features it exposes to a tester might be very useful if implemented in the VETS tool.

Test Report Tool The test report tool, like the VETS tool, is a minimal implementation. Many individual improvement of the tool could be envisioned,

which would increase its usability in general. This includes features such as test history, image scaling, test comparisons, better test description etc.

Especially interesting areas of further work on the tool would most likely include, a better utilization of the information provided by the image metrics component. Such utilization could include the possibilities for setting thresholds for particular images, further analysis by clicking specific metric type etc. This would help further decrease the dependence of a human tester to inspect every image in detail. Secondly, another feature could be better integration with the VETS tool and other software tools. The ability to invoke a test case with the human oracle procedure looking at the actual execution, in the case a test reports shows a failed correctness test, could be practical. Integration with tools like CCNET for sending summarized reports of tests could also be a practical feature. Finally, when the Image Collector is used, a feature which could group images in the collections according to the Platonic solid used, could help utilize the Image Collector in a more rational way.

User Feedback If the tools, and in particular the report tool, were developed further, so they were ready for test use in an more industrial environment, it would be very interesting to deploy the tools to get feedback from professional users.

Gathering input and feedback from programmers and testers working with applications that utilize graphical virtual environments, would be an important step towards improving the test tool. Letting people from the industry use the tool in startup projects as well as existing development projects, and receive response about their experiences, would be extremely useful, for evaluating the relevance and performance of both the procedure and the tools.

7.3 Conclusion

This section summarizes the conclusion from the previous chapters, and presents a final overall conclusion on the whole project, both procedure development and tool development, the results in relation to the problem analysis, and finally on the project process in general.

7.3.1 Tool Development

The following lists the sections of the individual conclusion found in this report, and gives a summarization of the conclusion:

- The VETS tool - section 4.4.3: The VETS tools was created to support the testing procedure created in this project. It was tested with two sample applications, and with the NAnt script tool, and found to be efficient as means of supporting the automation of testing process.

- Using AAR - section 4.5.3: The AAR functionality of the GM framework was investigated and tested for inclusion in the support tools developed. While AAR possess several a number of qualities, it is ultimately found not to be suitable for inclusion in the tools developed.

- The Image Collector - section 5.4: The Image Collector was implemented to support collection of multiple scene images in a virtual environment. It has been connected to the Delta3D class library, which makes it possible to get multiple scene views in Delta3D applications. The module has been tested in different Delta3D applications and was found to add significant value to image capturing and image rendering in these applications.

- The Image Analyzer - section 6.4.4: The Image Analyzer was implemented to compare collected scene images with stored references, and estimate the similarity between these scenes. The Image Analyzer has been used to analyze images from two Delta3D applications. Based on this, the conclusion is that programmers and testers might find it difficult to judge whether a test should pass or fail based on the metric calculations made by the image analyzer. However, the ability to visualize the scene differences has proven to be of great use in deciding whether tests should pass or fail.

- The Report Generator - section 6.5.3: The Report Generator is implemented in a simple form, and integrated with the Image Analysis tool to showcase the usefulness of the Report Generator. The tool is found to be a significant advantage, and is an obvious target of future work, to implement additional functionality.

7.3.2 Problem Analysis

Apart from the overall goal of developing a testing procedure for virtual environments, the problem analysis list three specific goals.

1. The implementation of image processing and analysis tool, and evaluation of these tools. In this project we have developed the Image Collector component, which is based on literature describing how to collect images for scene comparisons in virtual environments in an optimal way. To utilize this tool the Image Analyzer component was developed which could provide metrics on scene captures. Both tools were tested and evaluated on a developed application, and used in the overall test procedure.

2. The use of scene graphs as oracle information was briefly considered, and found to be infeasible in this project. The use of scene graphs can not be completely abandoned in test development, but where outside of the scope of this project, where development of framework incremental components was a goal. Working with scene graphs requires dedicated work with the OSG framework, and possibly architectural changes to the Delta3D framework.

3. Apart from the Image Collector component and the Image Analyzer component, the VETS tool was designed, implemented and tested as a support tool for the partial automation of the testing procedures. The tools was found to both usable and practical in the testing procedure, and we were able to integrate and combine the tool in an existing testing tool, in the form of NAnt.

The overall goal stated in the problem analysis was: “to develop an automated test tool”. We consider this goal to be fulfilled, and with reasonable success.

While we have not developed an fully automated tool, we have succeeded on singling out the most resource demanding steps in a fully manual testing procedure, and also in assisting these steps with our tools, to enable a semi-automated test procedure to take place. In addition we have described the necessary steps to be taken by a tester when working with virtual environments, in such a way that these steps may be used in any environment, supported by tools developed for the particular environment.

7.3.3 In General

We feel that the goals originally set out to be meet in this project has been reached, and that the they have been addressed in adequate detail during the work on this thesis. By this we mean that we have analyzed the problems concerning testing graphical virtual environments throughly, and that relevant areas of the domain has been considered. We have looked into what must be done to alleviate these problems, have developed tools that emphasis the solutions found.

The procedure found and the tools developed in this solution are sufficient to reach the goals set for this project. While this is true, there is still much work that can be done on both parts. Investigation into state-of-the-art knowledge in the area of regression testing in virtual environments, has revealed that there is very little information or tools available for the subject, neither commercial or scientific, and we feel that there is still much work to be done in the field in general. This project has just touched on the surface of the whole field of graphical regression testing of virtual environments in its analysis, and the procedure and tools are only a small step in experimenting with addressing this lack of general knowledge.

Appendix A

Installation and Configuration Guide

This chapter describes the installation and configuration of various tools, libraries and frameworks used in the project. This is not a review of the particular technologies, or a discussion of the different technologies particular suitability in different environments. Even though some of the information in the guide is of a general nature, the guide is written with the specific tools and settings in mind that are used throughout the project. No tests of any other configuration has been carried out, and there is no guaranty that instructions in the guide work for any other configurations of technologies.

A.1 Delta3D And Visual Studio

During the initial investigation of the content and scope of this thesis a number of tools were used. Among them the Delta3D game- and simulation framework was of special interest, since this framework constitutes the foundation of the software created in the thesis. In the preliminary work carried out to determine the scope and direction of this thesis, a considerable amount of time and work was used to set up a proper development environment in Delta3D. During that phase a number of tools was investigated, and for those considered relevant various experiments was performed. Since this work, the Delta3D framework has been released in a new version, namely version 2.1, which besides new features, contains major design changes and bug fixes. Because of this update, it was decided to migrate from Delta3D 1.5 to version 2.1.

The following explains how to install and build Delta3D 2.1 on a Windows XP Pro system with Service Pack 2 using Microsoft Visual Studio .NET 2008 (9.0) with Service Pack 1 as Integrated Development Environment (IDE).

At the time of writing this, Delta3D is available in version 2.1 and Microsoft has released versions 2008 (9.0) of Visual Studio since the preliminary work was carried out. Using the most resent releases of both products would be preferred, especially with respect to introducing a structured test process in a existing development project, as more recent versions of Visual Studio provides mechanisms to refactor code, which is a highly prioritized objective in TDD. Using the most recent versions of Visual Studio and Delta3D from the beginning

would have been preferable. However, this was not possible, since our collaboration partner, IFAD, used more outdated versions of these tools. At that time, porting to a more recent development platform was considered too risky due to the size of the code base of the IFACTS project. Additionally, we decided that working on the exact same platform as IFAD would be easier that converting to a previous platform at a later stage.

The Delta3D installation is available from [10]. The Windows installation of Delta3D can be performed by using on of the following distributions:

1. Windows pre-compiled installer including a setup guide.
2. Windows pre-compiled zip package.
3. The C++ source files.

By using the pre-compiled installer a lot of setup is performed automatically which is preferred. The following describes this approach. Details of installing Delta3D on other distributions and on other platforms is available from the Wiki page of Delta3D [11].

A.1.1 Compiling And Building Delta3D

To compile Delta3D it is necessary to first create a solution file for an IDE (or some other environment) and then build the project using that tool. Delta3D 2.1 does not come with pre-created VS2005/VS2008 solution files like older versions of Delta3D did.

To create a VS2008 solution for Delta3D 2.1.0 first obtain the following programs/resources:

1. Download Delta3D 2.1 from SourceForge.
2. Download Delta3D 2.1 dependencies from SourceForge.
3. Download the open source C++ edition of Qt from Trolltech.
4. Download the NMake build tool from Microsoft.
5. Download the CMake build tool from cmake.org.

Now the resources needed to start the build process are in place. It should be pointed out that the version of Qt must be the source code (src) version, not the MinGW version, which Delta3D can not use. The following contains a brief explanation of the process involved in installing and configuring Qt, NMake, CMake, and the other tools needed to build and compile Delta3D. It is assumed that VS2008 already has been installed.

1. Install NMake - create a directory for NMake, for example C:\Program Files\NMake and move the NMake executable file to the new folder. Run the file and it will unzip to a couple of files, among them nmake.exe.
2. Install CMake by following the instruction giving by the installer.
3. Unzip the Qt package to a destination directory, for example C:\Qt\4.4.3 (depending on version).

4. Install Delta3D from the executable - again just follow the instructions.

5. Unzip the Delta3D dependencies package to your Delta3D installation folder, for example C:\Program Files\Delta3D_REL-2.1.0. The package contains a single folder, ext, which should overwrite the existing Delta3D ext.

Now the tools are basically installed. The first step is then to compile Qt:

1. Add NMake to the environment path - For example add C:\Program Files\NMake to the path variable if that is where the NMake tool was installed.

2. Add the Qt bin folder to the path - if Qt was installed to C:\Qt\4.4.3 then add C:\Qt\4.4.3\bin to the path.

3. Start a Visual Studio 2008 Command Prompt, not the usual cmd, but the one embedded in VS2008 so that Qt can find the compilation (cl) and linking (link) tools.

4. Navigate to the Qt folder, for example C:\Qt\4.4.3, and run "configure.exe -platform win32-msvc-2008", which should create a file called "Makefile" in the Qt directory.

5. Run NMake by typing "nmake" and wait while Qt compiles. This takes a while.

Now that Qt is installed it should be possible to create a VS2008 solution file for Delta3D:

1. Open the CMake tool and select the CMake entry in the start menu (not the cmake-gui).

2. Open an explorer window and drag the CMakeLists.txt file from the Delta3D folder into the CMake window.

3. Click configure and wait while it checks your system. If it complains about QMAKE missing, find the relevant entry (some where in the bottom of the configurations in CMake) and point it to the qmake.exe in your Qt bin directory, and press configure again.

4. When successfully configured press OK.

Now a VS2008 solution file should have been created in the root of the Delta3D folder. Open that with Visual Studio and build the solution.

A.1.2 Delta3D Development Environment In Visual Studio

To use Delta3D in a custom Visual Studio project, it is necessary to setup a development environment. The following show the details involved in setting up various environmental variables in Visual Studio.

Include Files For A Delta3D Project In Visual Studio

The installation of Delta3D has provided a number of environmental variables. To use the **$(DELTA_INC)** variable in Visual Studio:

1. Right click on the given project in the Visual Studio Solution Explorer and select "Properties".
2. Select "C/C++". Different sub-categories appear. Select "General" and change the field marked "Additional Include Directories" to **$(DELTA_INC)**. Figure A.1 shows the setup window after performing these steps.

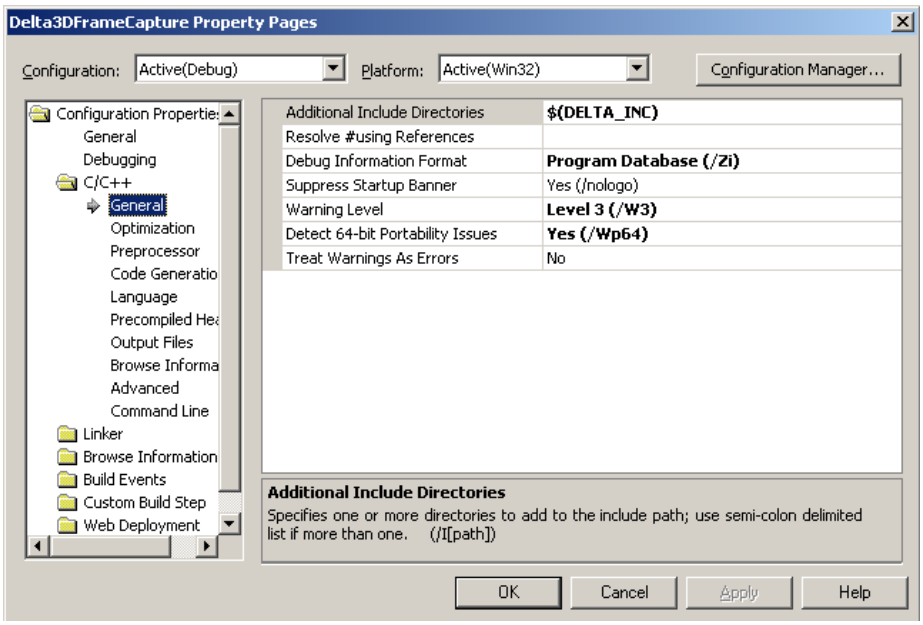


Figure A.1: Setting up "Additional Include Directories" in Visual Studio.

3. In the "C/C++" category, select the sub-category "Code Generation" and change "Runtime Library" to: **Multi-threaded Debug DLL (/MDd)** as shown in figure A.2.
4. In the "C/C++" category, select the sub-category "Precompiled Headers" and set the field "Create/Use Precompiled Header" to: **Not using Precompiled Headers**. Figure A.3 shows the setup windows after preforming this step.

Library Files For A Delta3D Project In Visual Studio

Another environmental variable created during the installation of Delta3D, is the **$(DELTA_LIB)** variable. To use this variable in Visual Studio the following steps must be carried out.

1. Right click on the given project in the Visual Studio Solution Explorer and select "Properties".

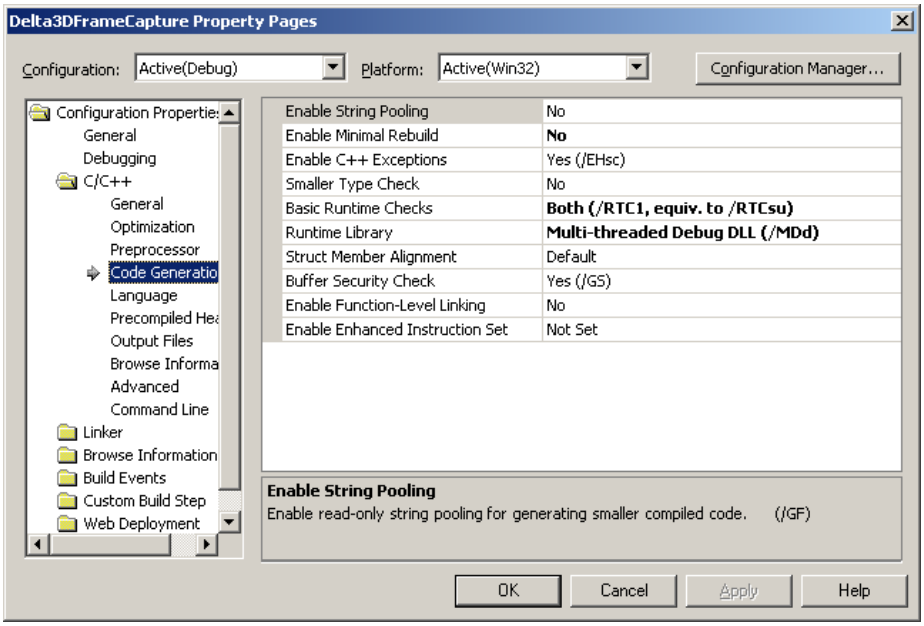


Figure A.2: Setting up "Runtime Library" in Visual Studio.

2. In the "Linker" category, select the sub-category "General" and set the field "Additional Library Directories" to: **$(DELTA_LIB)** as shown in figure A.4.

3. In the "Linker" category, select the sub-category "Input" and click the button to the right of the entry marked "Additional Dependencies". This is show in figure A.5.

4. Add the entries shown in figure A.6 in the input box:

A.2 OpenSceneGraph And Visual Studio

OpenSceneGraph (OSG) is part of the Delta3D framework as an external dependency, but unfortunately part of the OSG library included in Delta3D 2.1 is compiled with Visual Studio 2005. This introduces conflicts when Delta3D, and implicitly OSG, have been compiled with Visual Studio 2008, since the dynamic link library (DLL) files therefore are incompatible.

To overcome these issues OSG have to be recompiled with Visual Studio to make the DLLs compatible. The following describes this process.

1. Get OSG:
   (a) Download an OSG version from the SVN-trunk [12].
   (b) Download the OSG-dependencies from [13].
   (c) Place the OSG-dependencies in OSG\ext.

2. Build OSG:

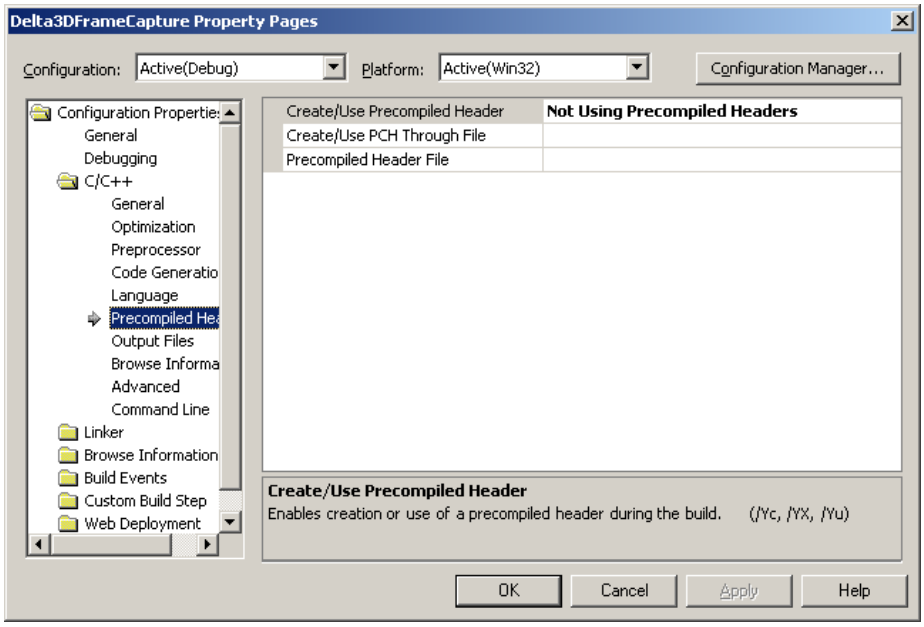


Figure A.3: Setting up "Precompiled Headers" in Visual Studio.

(a) Run CMake with CMakeLists.txt (included in the OSG root folder).

(b) Set ACTUAL_3DPARTY_DIR to the OSG-dependencies folder.

(c) Set GDAL_INCLUDE_DIR to the Delta3D dependencies folder\inc or wherever GDAL was installed.

(d) Set GDAL_LIBRARY to the Delta3D dependencies folder\lib or wherever GDAL was installed.

(e) Activate "Show Advanced Entries" and check if all freetype-include paths are set to the OSG-dependencies folder\include.

3. Set environment variables:

(a) Set OSG_ROOT = the OSG-folder.

(b) Set OSG_FILE_PATH = "%OSG_ROOT%\data" if the OSG datasets have been downloaded.

(c) Set OSG_EXT = the OSG-dependencies folder.

(d) Add to PATH: ";%OSG_ROOT%\bin;%OSG_EXT%\bin".

(e) Relogin.

4. Prepare Delta3D-build:

(a) Delete all OSG, openthreads, libpng, and zlib related files in the Delta3D-dependencies folder to avoid DLL conflicts and make future OSG updates more comfortable.

(b) To take advantages of the updated png-library from Delta3D, copy osgdb_pngd.dll from the current Delta3D-dependencies into the OSG-folder\bin and move libpng12.dll and libpng12d.dll in the OSG-dependencies folder\bin.

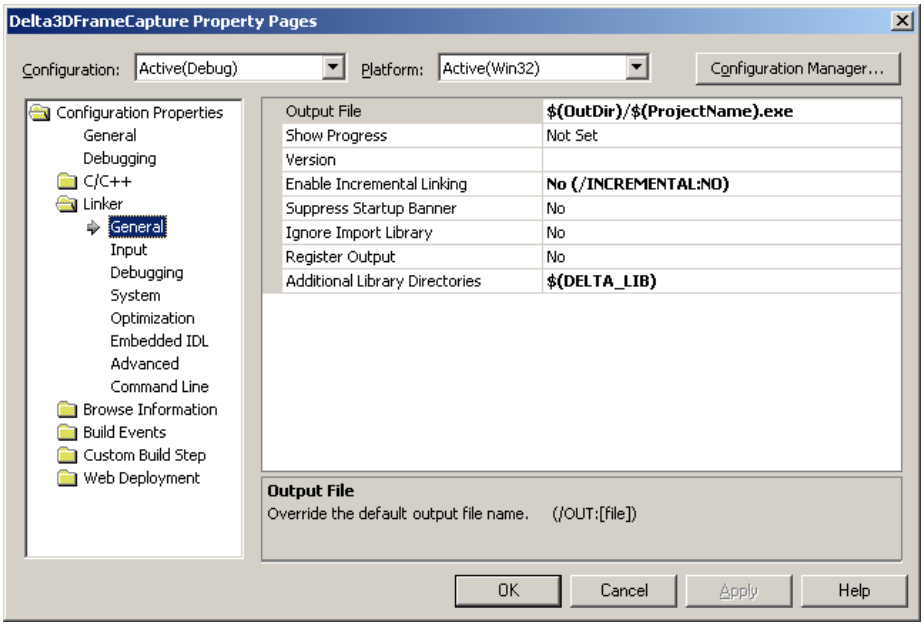


Figure A.4: Setting up "Additional Library Directories" in Visual Studio.

5. Build Delta3D:

   (a) Run CMake again with Delta3D CMakeList.txt like explained above.

   (b) All OSG related paths and files should be found automatically.

A.3 Configuring Cruise Control .NET for C++

The Cruise Control .NET (CCNET) automatic build server [2] is highly configurable, and can be adapted for a plethora of different development tools. Unfortunately this also means that the server configuration files are complex and has a huge amount possible settings. In the following a guide for using CCNET with a Visual Studio 2003 C++ project and NAnt is presented.

A.3.1 Cruise Control .NET

CCNET can be installed on any Microsoft Windows machine, in this example a Windows XP Professional installation. The default installation options for CCNET are appropriate, and no special settings are required. The CCNET core requirements are limited to the Microsoft .NET 1.0 or 1.1 framework, but to get the full advantage of the build server, an ASP.NET capable web server (like Microsoft Internet Information Server (ISS)) is required. In general CCNET consists of three components which is shown in table A.1 along with their requirements. The CCNET installation folder,called **%ccnetfolder%**, contains three folders: *Examples*, *server* and *webdashboard*. The *server* folder contains the configuration file, *ccnet.config*, that should be edited, while other examples of configuration are found in the *Examples* folder.

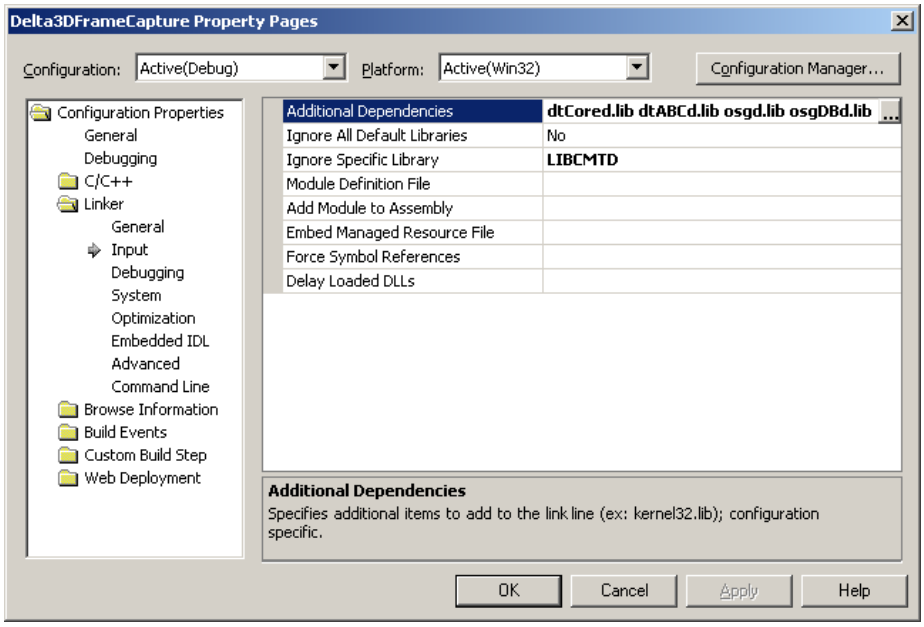


Figure A.5: Setting up "Additional Dependencies" in Visual Studio.

Component	Prerequisites
CruiseControl.NET Server	Microsoft.NET Framework Version 1.0 or 1.1
CruiseControl.NET Web Application	- Microsoft.NET Framework Version 1.0 or 1.1 - ASP.NET-enabled web server (typically IIS with ASP.NET configured)
Developer desktop running CCTray	Microsoft.NET Framework Version 1.0 or 1.1

Table A.1: CruiseControl.NET installation prerequisites

The main configuration of the build server is found in the %ccnetfolder%/server folder in the file **ccnet.config**. The format is XML style and allows configuration of multiple projects each with multiple tasks[1] in on single file. A complete overview of the many configuration options are found in [14].

Running CCNET in a console

If CCNET is run from a console, the easiest way of setting up the environment is to execute the vsvars32.bat script. This file imports all environment variables from VS2003 to the current environment. The script can be executed for each new terminal, or its is possible to create a script file as shown in listing A.1.

```
@echo off
@call "C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools\vsvars32.bat"
"C:\Program Files\nant\bin\nant.exe" %*
```

Listing A.1: NAnt script with VS2003 ENV

This batch file imports the VS2003 environment before executing the NAnt executable. To run the scrip automatically, point this CCNET configuration

[1] like several different builds, unit testing etc

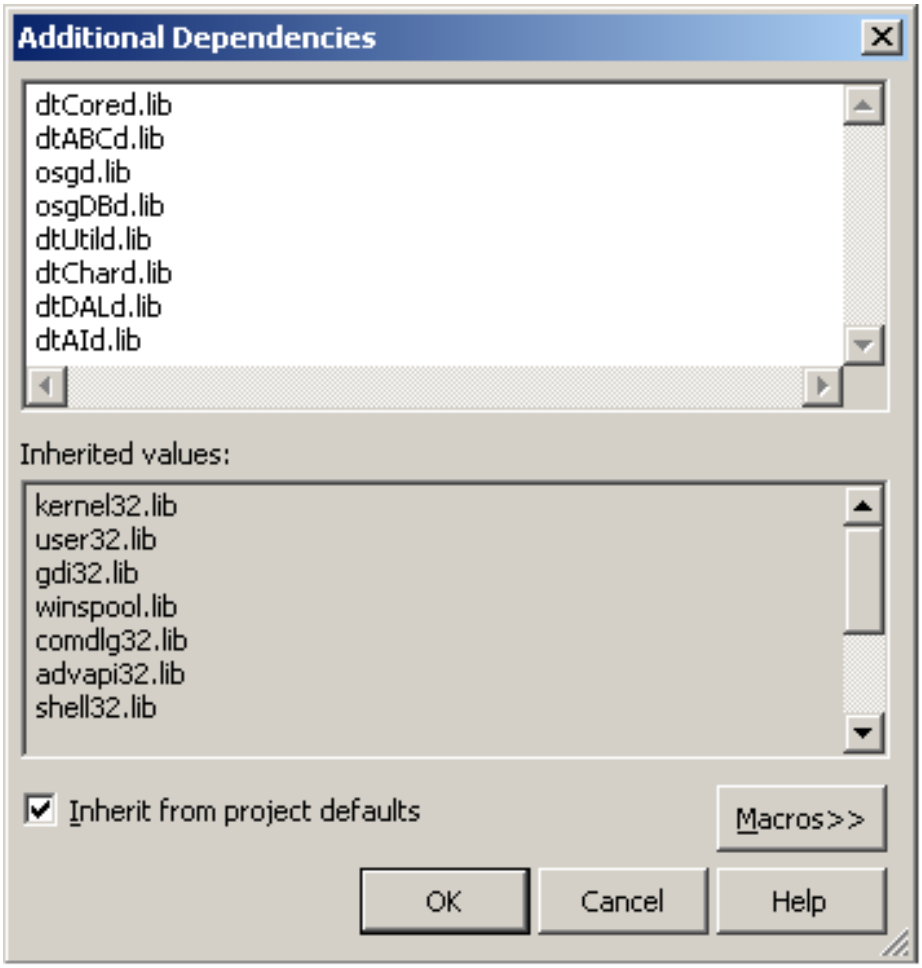


Figure A.6: Including link libraries in Visual Studio.

to the batch file, rather than directly to the executable. It is also possible to execute the vsvars32.bat script by changing the registry database, both solution are described by [15].

Running CCNET as a service

To ensure that the CCNET is running always, the server can be added as a service on the machine where it's installed. The default installation includes an executable file that can be used when run as a service. To add CCNET as a service follow the instructions given by [16]. It is important to notice that the user account running the service must have access to the SVN server, unless the CCNET configuration file specifically gives username and password to be used with SVN. As with the console application, the VS2003 environment variables must be exported.

Cruise Control .NET Tools

The CCNET installation includes two non-core components:

- the web dashboard
- the CCTray application

The web dashboard can be installed with default settings. The %ccnetfolder%/webdashboard is automatically linked to the IIS installation on the machine, and the dashboard can immediately be accessed at **http://localhost/ccnet**. Figure A.7 shows the web dashboard used in this project running on the server *tek-4116*. Three build projects are shown on the dashboard, all with a successful last build status. The

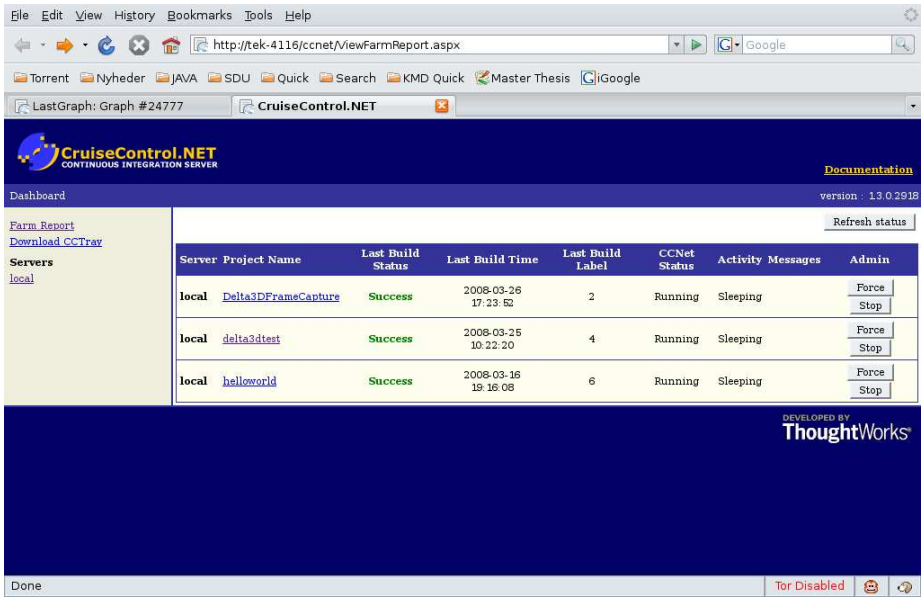


Figure A.7: The Cruise Control .NET web dashboard.

web dashboard provides access to the latest build status of several projects on the overview page, as well as more detailed information about builds and errors on the specific project pages. This is all setup by default, event though build report formats etc. can be configured if necessary.

The CCTray application is a stand alone application that is able to run on each developers machine. It polls the build server for project details, and provides a small task-bar icon (in Windows) that shows the current project status, and gives notifications on important events, like a broken build or a repaired build. Figure A.8 shows the CCTray application main window. Projects must be added manually to the tray for monitoring, but the addition of a build server and projects are straight forward. The application supports several types of connections, but uses a Microsoft Remoting connection by default, which is likely to only work on a local network.

A.3.2 NAnt

NAnt [3] is described on the NAnt website:

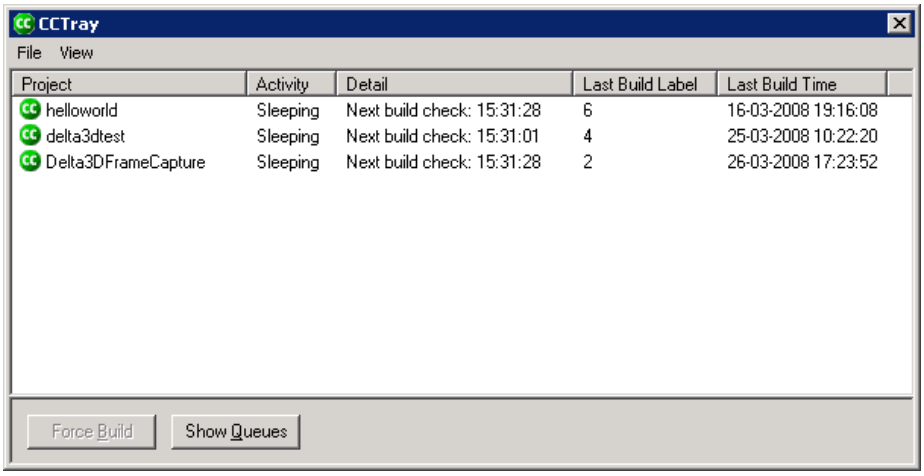


Figure A.8: The Cruise Control .NET CCTray main window.

> NAnt is a free .NET build tool. In theory it is kind of like make without make's wrinkles. In practice it's a lot like Ant.

Basically NAnt is used to build a project, by setting up the appropriate environment variables and then linking and compiling the required libraries in the project in the correct order. NAnt is a stand-alone tool, but can be integrated in an IDE like VS2003, similar to the way Eclipse uses Ant as build tool, and VS2005 uses MSBuild[17].

NAnt can be integrated in Visual Studio 2003, rather that the default build system, and can by default use a VS2003 solution file as target. This means that no additional work is needed when setting up an existing VS2003 project for NAnt, the NAnt tool can simply use the solution file.

Pre-pequisites for installing NAnt is limited to any version of Microsofts .NET frameworks or a compatible framework, like Mono. NAnt is packages as a ZIP file, and can be unpacked to any appropriate location. The NAnt installation folder will be called **%nantfolder%** in this guide.

Integrating NAnt with Visual Studio 2003

1. Create a new External Tool by going to Tools — External Tools
2. Click Add to add a new tool
3. Give it a title of "NAnt"
4. Browse to the location of the NAnt.exe file wherever you have it downloaded to
5. Set the initial directory to $(SolutionDir) (where your .build file resides)
6. Click on "Use Output Window"
7. Click OK

To override the shortcut **Ctrl+Shift+b**, so that it will run the NAnt tool, the following can be done:

1. Go to the Options dialog by going to Tools — Options
2. Under the Environment node in the tree click on Keyboard
3. Find the command called "Tools.ExternalCommand1" by entering it the "Show commands containing:" text box or scrolling through the list
4. Click on the "Press shortcut key(s):" text box
5. Press Ctrl+Shift+B
6. You'll see it's already assigned to the "Build.BuildSolution" command but click on the "Assign" button to reassign it
7. Click OK to close the dialog.

Additional NAnt integration support for VS2003 may be found in [18] where some advice on how to create support for NAnt configuration files in VS2003 is given. For a good beginners introduction to NAnt configurations [19] is a very good resource. In this project a very simple build file is used, formatted as shown in listing A.2.

```
1  <?xml version="1.0"?>
2  <project default="run">
3    <target name="run" depends="clean">
4      <solution solutionfile="Delta3DFrameCapture.sln"  ↩
           configuration="Debug" />
5    </target>
6    <target name="clean">
7      <delete dir="Source" failonerror="false" />
8      <mkdir dir="Source" />
9    </target>
10 </project>
```

Listing A.2: NAnt config file

In this configuration NAnt is simply pointed directly to the .sln solution file. Note that with this setup there is no immediate way of specifying the specific target to be run, the default is always chosen.

Appendix B

Spatial Transformations in Delta3D

This chapter describes the mathematics behind camera transformations in 3D virtual environments. The point of reference for this formulation is how object positions and rotations are managed in the Delta3D framework, but the concepts in this derivation can easily be applied to other graphical engines.

Scene manipulation, by definition, implies that objects and cameras will be moved around in virtual space. This leads to the need of representing positions and orientations of the various entities contained in the scene. The following section describes the mathematics behind spatial transformations in Delta3D. The focus in this thesis is not designing and creating a 3D graphics engine, but rather how an existing graphical framework engine can be extended to include support for validation and verification of the graphics contained in an virtual environment. Therefore the material presented in the following could seem misplaced, but since the software developed in this thesis makes extensive use of the existing spatial transformations already included in Delta3D, it was decided to include the relevant mathematical formulation of how spatial transformations are handled by the framework. Finally, another argument for including this material is that the documentation of Delta3D totally lacks information on how the framework performs transformations. Therefore the material presented in the following is collected by studying the relevant source code files in Delta3D. The mathematical derivation follows the principles described in [Craig, 1989] and [Slabaugh, 2008].

B.1 Position

The location of a point P in virtual space is specified by a 3×1 position vector. Figure B.1 pictorially represents a Delta3D coordinate system, with three mutually orthogonal unit vectors X, Y, and Z. A point P is represented with a vector and can equivalently be thought of as a position in space. Individual elements of the position vector have subscripts x, y, and z. The position vector is defined as:

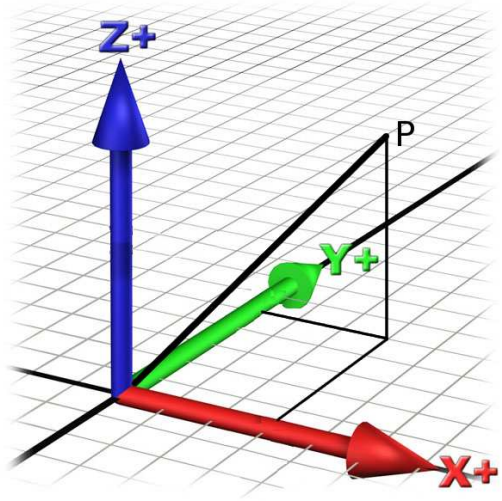


Figure B.1: A position vector in Delta3D.

$$\boldsymbol{P} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}, \tag{B.1}$$

where p_x, p_y, and p_z are components of P with numerical values which indicate distances along the corresponding axes. Each of these distances along an axis are the result of projecting the vector onto the corresponding axis.

B.2 Rotation

The location of a point in a virtual environment can be specified by the position vector given in equation B.1. For a point the position vector is sufficient to describe the location of the point in space. However, when considering more complex objects the orientation needs to be taken into account in order to completely describe the location of the object in space.

Figure B.2 shows the default coordinate system in Delta3D. A position is specified in Cartesian (x, y, z) coordinates, where $+X$ is to the right, $+Y$ is forward into the screen, and $+Z$ is up. A rotation is specified by heading, pitch, and roll (h, p, r) - all in degrees, where heading is rotation about the Z axis, pitch is rotation about the X axis, and roll is rotation about the Y axis. The angles of rotation all obey the right-hand-rule, which can be seen in the figure. Since a rotation involves rotating a point about a maximum of three axes, the order of axis rotation matters. Therefore rotation in Delta3D occurs in the order Z, X, and Y also known as Z-X-Y Euler angles. The rotation matrix for rotating an object an angle p (pitch) about the X-axis is given by equation B.2:

$$\boldsymbol{R}_x(p) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(p) & \sin(p) \\ 0 & -\sin(p) & \cos(p) \end{bmatrix}. \tag{B.2}$$

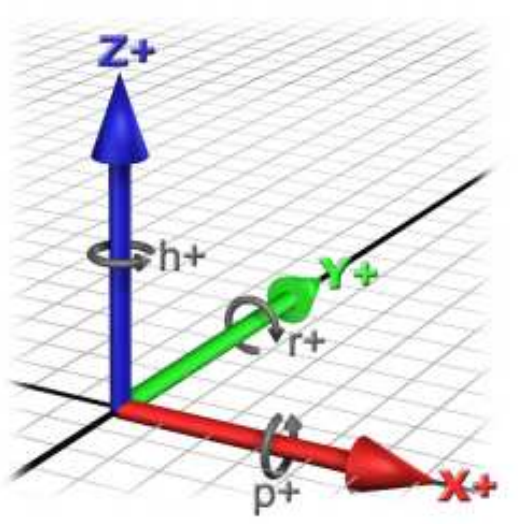


Figure B.2: Default coordinate system in Delta3D.

The rotation matrix for rotating an object an angle r (roll) about the Y-axis is given by equation B.3:

$$\boldsymbol{R}_y(r) = \begin{bmatrix} \cos(r) & 0 & -\sin(r) \\ 0 & 1 & 0 \\ \sin(r) & 0 & \cos(r) \end{bmatrix}. \tag{B.3}$$

The rotation matrix for rotating an object an angle h (heading) about the Z-axis is given by equation B.4:

$$\boldsymbol{R}_z(h) = \begin{bmatrix} \cos(h) & \sin(h) & 0 \\ -sin(h) & \cos(h) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{B.4}$$

To determine the rotation matrix of an object rotated about multiple axes, the rotation matrices are multiplied. Since matrix multiplication is not communicative the order of multiplication matters. The derivation of the equivalent rotation matrix $\boldsymbol{R}_{ZXY}(h,p,r)$ is straightforward and given by:

$$\begin{aligned} \boldsymbol{R}_{ZXY}(h,p,r) &= \boldsymbol{R}_y(r)\boldsymbol{R}_x(p)\boldsymbol{R}_z(h) \\ &= \begin{bmatrix} cr & 0 & -sr \\ 0 & 1 & 0 \\ sr & 0 & cr \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cp & sp \\ 0 & -sp & cp \end{bmatrix} \begin{bmatrix} ch & sh & 0 \\ -sh & ch & 0 \\ 0 & 0 & 1 \end{bmatrix}, \end{aligned} \tag{B.5}$$

where ch is shorthand for $\cos(h)$ and sh for $\sin(h)$, etc. It is important to understand the order of rotations used in equation B.5. Thinking in terms of rotations as operators, the rotations have been applied (from the right) $\boldsymbol{R}_z(h)$, then $\boldsymbol{R}_x(p)$, and then $\boldsymbol{R}_y(r)$. Performing the matrix multiplications give the rotation matrix:

$$\boldsymbol{R}_{ZXY}(h,p,r) = \begin{bmatrix} chcr - shsrsp & crsh + srspch - srcp & -srcp \\ -shcp & chcp & sp \\ srch + shcrsp & srsh - chspcr & crcp \end{bmatrix}. \tag{B.6}$$

Equation B.6 is the rotation matrix used in Delta3D and is valid for rotations performed in the order: about Z by h, about X by p, and about Y by r. The Delta3D class `dtUtil::MatrixUtil` implements equation B.6 in the method `HprToMatrix()`.

B.3 Euler Angles from Rotation Matrix

The inverse problem, that of extracting equivalent Z-X-Y fixed angles from the rotation matrix is often of interest. The solution depends on solving a set of transcendental equations. The derivation is done by equating each element in $\boldsymbol{R}_{ZXY}(h,p,r)$ with the corresponding element in the matrix product $\boldsymbol{R}_z(h)\boldsymbol{R}_{\boldsymbol{x}}(p)\boldsymbol{R}_y(r)$. Let

$$\boldsymbol{R}_{ZXY}(h,p,r) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}. \tag{B.7}$$

From equation B.6 it can be seen that

$$r_{23} = \sin(p). \tag{B.8}$$

Taking the inverse sinus function on both sides of the equation yields the angle p:

$$p = \sin^{-1}(r_{23}) \tag{B.9}$$

However, one must be careful in interpreting this equation. Since $\sin(\pi - p) = \sin(p)$, there are actually two distinct values (for $r_{23} \neq \pm 1$) of p that satisfy equation B.9. Therefore, both the values:

$$p_1 = \sin^{-1}(r_{23}) \tag{B.10}$$

$$p_2 = \pi - p_1 = \pi - \sin^{-1}(r_{23}) \tag{B.11}$$

are valid solutions. The special case of $r_{23} = \pm 1$ is handled later in this report. By using the r_{23} element of the rotation matrix, two possible values can be determined.

B.4 Finding the corresponding angles of h

To find the values for h the following relationship is used:

$$\frac{-r_{21}}{r_{22}} = \tan(h). \tag{B.12}$$

Solving for h gives

$$h = \operatorname{atan2}(-r_{21}, r_{22}), \tag{B.13}$$

where $\operatorname{atan2}(y,x)$ is arc tangent of the two variables x and y. It is similar to calculating the arc tangent of y/x, except that the signs of both arguments are used to determine the quadrant of the result, which lies in the range $[-\pi, \pi]$. However, if $\cos(p) > 0$ equation B.13 holds, but when $\cos(p) < 0$, $h = \operatorname{atan2}(r_{21}, -r_{22})$. A way to handle this is to use the equation

$$h = \operatorname{atan2}\Big(\frac{r_{21}}{\cos(p)}, \frac{r_{22}}{\cos(p)}\Big) \tag{B.14}$$

to compute h. Equation B.14 is valid for all cases, except when $\cos(p) = 0$. Again, this special case is handled later. For each value of p, the corresponding value of h is calculated by equation B.14, yielding

$$h_1 = \text{atan2}\Big(\frac{r_{21}}{\cos(p_1)}, \frac{r_{22}}{\cos(p_1)}\Big) \tag{B.15}$$

$$h_2 = \text{atan2}\Big(\frac{r_{21}}{\cos(p_2)}, \frac{r_{22}}{\cos(p_2)}\Big). \tag{B.16}$$

B.5 Finding the corresponding angles of r

A similar analysis holds for finding r. From the rotation matrix the following relationship holds:

$$\frac{-r_{13}}{r_{33}} = \tan(r). \tag{B.17}$$

Solving for r gives

$$r = \text{atan2}\Big(\frac{r_{13}}{\cos(p)}, \frac{r_{33}}{\cos(p)}\Big). \tag{B.18}$$

Again, this equation is valid for all cases, except when $\cos(p) = 0$. For each value of p, the corresponding value of r is calculated using equation B.18, which gives

$$r_1 = \text{atan2}\Big(\frac{r_{13}}{\cos(p_1)}, \frac{r_{33}}{\cos(p_1)}\Big) \tag{B.19}$$

$$r_2 = \text{atan2}\Big(\frac{r_{13}}{\cos(p_2)}, \frac{r_{33}}{\cos(p_2)}\Big). \tag{B.20}$$

B.6 The two solutions

For the case of $\cos(p) \neq 0$, there exist two triplets of Euler angles that reproduce the rotation matrix, namely (h_1, p_1, r_1) and (h_2, p_2, r_2). Both of these solutions will be valid and they correspond to rotate either clockwise or counterclockwise.

B.7 Special case

The technique described above does not work if the r_{23} element of the rotation matrix is 1 or -1, which corresponds to $p = -\pi/2$ or $p = \pi/2$ respectively. This happens because $\cos(p) = 0$ in these situations, which causes problems regarding determining the angles h and r because attempting to divide by zero occurs in equations B.14 and B.18. In this case, r_{13}, r_{21}, r_{22}, and r_{33} do not constrain the values of h and r. Therefore, different elements of the rotation matrix must be used in order to compute the values of h and r.

If $p = \pi/2$:

$$r_{11} = \cos(h)\cos(r) - \sin(h)\sin(r) = \cos(h + r),$$

$$r_{12} = \cos(r)\sin(h) + \sin(r)\cos(h) = \sin(h + r),$$

$$r_{31} = \sin(r)\cos(h) + \sin(h)\cos(r) = \sin(h + r) = r_{12},$$

$$r_{32} = \sin(r)\sin(h) - \cos(h)\cos(r) = -\cos(h + r) = -r_{11}.$$

Any h and r that satisfy these equations will be a valid solution. Using the equations for r_{11} and r_{12}, give

$$(h + r) = \text{atan2}(r_{11}, r_{12}),$$

$$h = -r + \text{atan2}(r_{11}, r_{12}).$$

If $p = -\pi/2$:

$$r_{11} = \cos(h)\cos(r) + \sin(h)\sin(r) = \cos(h - r),$$

$$r_{12} = \cos(r)\sin(h) - \sin(r)\cos(h) = \sin(h - r),$$

$$r_{31} = \sin(r)\cos(h) - \sin(h)\cos(r) = -\sin(h - r) = -r_{12}$$

$$r_{32} = \sin(r)\sin(h) + \cos(h)\cos(r) = \cos(h - r) = r_{11}.$$

Again, using the equations for r_{11} and r_{12} the angle is given by

$$(h - r) = \text{atan2}(r_{11}, r_{12}),$$

$$h = r + \text{atan2}(r_{11}, r_{12}).$$

The above derivation shows that in both the $p = \pi/2$ and $p = -\pi/2$ cases h and r are linked. This phenomenon is called Gimbal lock [Slabaugh, 2008]. Although in this case, there are an infinite number of solutions to the problem. Most often only one solution is of interest. Finding a solution is therefore often done by setting $r = 0$ and computing $h = \text{atan2}(r_{11}, r_{12})$.

In Delta3D the h, p, r angles are obtain by calling the method `MatrixToHpr`() found in class `dtUtil::MatrixUtil`.

Appendix C

The Platonic Solids

In the following a brief introduction to the five Platonic solids is given, including the mathematics necessary for calculating the position of the vertices. The equations are all derived with respect to the Delta3D coordinate system described in section 4.2.3, with the origin placed at the center of the solids. The mathematics behind the following formulas are based on information found at MathWorld which is part of the Wolfram website [1].

C.1 Tetrahedron

The tetrahedron is a solid geometry having four vertices, six edges, and four faces as shown in figure C.1 - create by Robert Webb and available from [20]. In

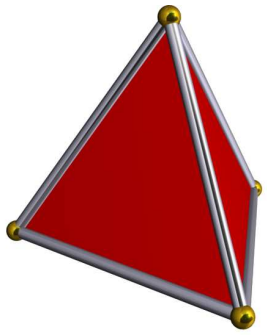


Figure C.1: The tetrahedron is one of the five Platonic solids with four vertices, six edges, and four faces.

order to place a camera at the vertices of the tetrahedron, the Cartesian (x, y, z) coordinates of the vertices must be calculated as a function of the edge length a and the center point c of the tetrahedron. These calculations are presented in the following. More details can be found in [21].

[1] http://mathworld.wolfram.com

Vertex V1: The position vector of the top vertex of the tetrahedron, denoted $\vec{V}_1$, has the following components.

$$\vec{V}_1 = \begin{bmatrix} x_c + x_1 \\ y_c + y_1 \\ z_c + z_1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c + \frac{\sqrt{6}}{4}a \end{bmatrix}. \tag{C.1}$$

Vertex V2: The position vector of the front lower left vertex of the tetrahedron, denoted $\vec{V}_2$, has the following components.

$$\vec{V}_2 = \begin{bmatrix} x_c + x_2 \\ y_c + y_2 \\ z_c + z_2 \end{bmatrix} = \begin{bmatrix} x_c - \frac{1}{2}a \\ y_c - \frac{\sqrt{3}}{6}a \\ z_c - \frac{\sqrt{6}}{12}a \end{bmatrix}. \tag{C.2}$$

Vertex V3: The position vector of the back lower center vertex of the tetrahedron, denoted $\vec{V}_3$, has the following components.

$$\vec{V}_3 = \begin{bmatrix} x_c + x_3 \\ y_c + y_3 \\ z_c + z_3 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c + \frac{\sqrt{3}}{3}a \\ z_c - \frac{\sqrt{6}}{12}a \end{bmatrix}. \tag{C.3}$$

Vertex V4: The position vector of the front lower right vertex of the tetrahedron, denoted $\vec{V}_4$, has the following components.

$$\vec{V}_4 = \begin{bmatrix} x_c + x_4 \\ y_c + y_4 \\ z_c + z_4 \end{bmatrix} = \begin{bmatrix} x_c + \frac{1}{2}a \\ y_c - \frac{\sqrt{3}}{6}a \\ z_c - \frac{\sqrt{6}}{12}a \end{bmatrix}. \tag{C.4}$$

C.2 Hexahedron

The hexahedron is a solid geometry composed of six square faces that meet each other at right angles. The hexahedron has eight vertices and twelve edges as shown in figure C.2 - create by Robert Webb and available from [20]. The equations describing the vertex positions of the hexahedron with respect to the center point c and the edge length a are rather simple. Therefore the following only shows the position of the front upper left vertex, since the equations describing the other vertex positions are almost equivalent - the only difference is the sign specifying the quadrant of the coordinate system. More details about the geometric properties of the hexahedron can be found in [22].

The position of the front upper left vertex of the hexahedron, denoted $\vec{V}_1$, has the following components.

$$\vec{V}_1 = \begin{bmatrix} x_c + x_1 \\ y_c + y_1 \\ z_c + z_1 \end{bmatrix} = \begin{bmatrix} x_c - \frac{1}{2}a \\ y_c - \frac{1}{2}a \\ z_c + \frac{1}{2}a \end{bmatrix}. \tag{C.5}$$

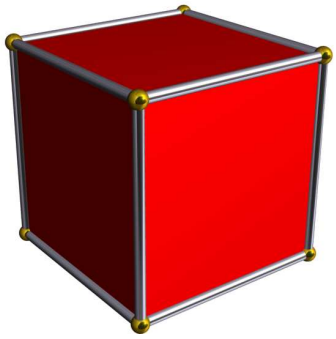


Figure C.2: The hexahedron is one of the five Platonic solids with eight vertices, twelve edges, and six faces.

C.3 Octahedron

The octahedron is a solid geometry having six vertices, twelve edges, and eight equivalent equilateral triangular faces as shown in C.3 - create by Robert Webb and available from [20]. The equations describing the positions of the vertices

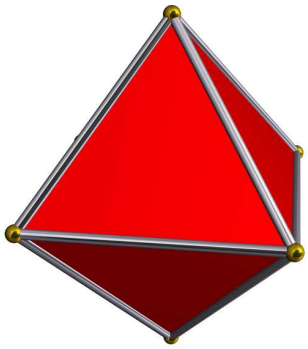


Figure C.3: The octahedron is one of the five Platonic solids with six vertices, twelve edges, and eight faces.

of the octahedron are presented in the following. A more thorough survey of the geometric properties of the octahedron can be found in [23].

Vertex V1: The position vector of the top vertex of the octahedron, denoted $\vec{V}_1$, has the following components.

$$\vec{V}_1 = \begin{bmatrix} x_c + x_1 \\ y_c + y_1 \\ z_c + z_1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c + \frac{\sqrt{2}}{2}a \end{bmatrix}. \tag{C.6}$$

Vertices V2, V3, V4, and V5: The positions of the four center vertices of the octahedron all lie in the corners of a square in the plane at $z = 0$. For

$i = \{2, 3, 4, 5\}$ the following equation describes the positions of the four center vertices.

$$\vec{V}_i = \begin{bmatrix} x_c + x_i \\ y_c + y_i \\ z_c + z_i \end{bmatrix} = \begin{bmatrix} x_c \pm \frac{1}{2}a \\ y_c \pm \frac{1}{2}a \\ z_c \end{bmatrix}, \tag{C.7}$$

where $+$ specifies the positive quadrant of the coordinate system, and $-$ is used if the vertices lie in the negative quadrant.

Vertex V6: The position vector of the bottom vertex of the octahedron, denoted $\vec{V}_6$, has the following components.

$$\vec{V}_6 = \begin{bmatrix} x_c + x_6 \\ y_c + y_6 \\ z_c + z_6 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c - \frac{\sqrt{2}}{2}a \end{bmatrix}. \tag{C.8}$$

C.4 Icosahedron

The icosahedron is a solid geometry having twelve vertices, thirty edges, and twenty equivalent equilateral triangular faces as shown in figure C.4 - create by Robert Webb and available from [20]. The equations describing the vertex

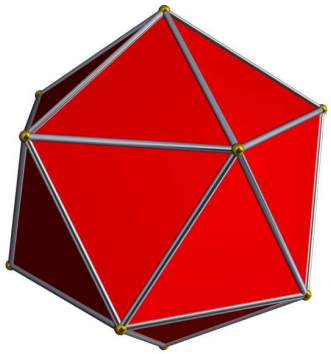


Figure C.4: The icosahedron is one of the five Platonic solids with twelve vertices, thirty edges, and twenty faces.

positions of the icosahedron are more complicated compared to the equations describing the positions of the vertices for the previous Platonic solids. Before presenting these equations a couple of useful constants and expressions are defined. Let $\phi = \frac{1}{2}(1 + \sqrt{5})$ and ratio $= \frac{\sqrt{10+2\sqrt{5}}}{4\phi}$. The radius R of the circumscribed sphere is a useful parameter in defining the position equations. The circumscribe radius is given by

$$R = \frac{\sqrt{10 + 2\sqrt{5}}}{4a} \tag{C.9}$$

From this the following two equations are defined

$$p = \frac{1}{2}\frac{R}{\text{ratio}} \tag{C.10}$$

and

$$q = 2\phi \frac{R}{\text{ratio}} \tag{C.11}$$

The twelve vertex positions can now be expressed be the following system

$$\vec{V}_1 = \begin{bmatrix} x_c + x_1 \\ y_c + y_1 \\ z_c + z_1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c + q \\ z_c - p \end{bmatrix}. \tag{C.12}$$

$$\vec{V}_2 = \begin{bmatrix} x_c + x_2 \\ y_c + y_2 \\ z_c + z_2 \end{bmatrix} = \begin{bmatrix} x_c + q \\ y_c + p \\ z_c \end{bmatrix}. \tag{C.13}$$

$$\vec{V}_3 = \begin{bmatrix} x_c + x_3 \\ y_c + y_3 \\ z_c + z_3 \end{bmatrix} = \begin{bmatrix} x_c - q \\ y_c + p \\ z_c \end{bmatrix}. \tag{C.14}$$

$$\vec{V}_4 = \begin{bmatrix} x_c + x_4 \\ y_c + y_4 \\ z_c + z_4 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c + q \\ z_c + p \end{bmatrix}. \tag{C.15}$$

$$\vec{V}_5 = \begin{bmatrix} x_c + x_5 \\ y_c + y_5 \\ z_c + z_5 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c - q \\ z_c + p \end{bmatrix}. \tag{C.16}$$

$$\vec{V}_6 = \begin{bmatrix} x_c + x_6 \\ y_c + y_6 \\ z_c + z_6 \end{bmatrix} = \begin{bmatrix} x_c - p \\ y_c \\ z_c + q \end{bmatrix}. \tag{C.17}$$

$$\vec{V}_7 = \begin{bmatrix} x_c + x_7 \\ y_c + y_7 \\ z_c + z_7 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c - q \\ z_c - p \end{bmatrix}. \tag{C.18}$$

$$\vec{V}_8 = \begin{bmatrix} x_c + x_8 \\ y_c + y_8 \\ z_c + z_8 \end{bmatrix} = \begin{bmatrix} x_c + p \\ y_c \\ z_c - q \end{bmatrix}. \tag{C.19}$$

$$\vec{V}_9 = \begin{bmatrix} x_c + x_9 \\ y_c + y_9 \\ z_c + z_9 \end{bmatrix} = \begin{bmatrix} x_c + p \\ y_c \\ z_c + q \end{bmatrix}. \tag{C.20}$$

$$\vec{V}_{10} = \begin{bmatrix} x_c + x_{10} \\ y_c + y_{10} \\ z_c + z_{10} \end{bmatrix} = \begin{bmatrix} x_c - p \\ y_c \\ z_c - q \end{bmatrix}. \tag{C.21}$$

$$\vec{V}_{11} = \begin{bmatrix} x_c + x_{11} \\ y_c + y_{11} \\ z_c + z_{11} \end{bmatrix} = \begin{bmatrix} x_c + q \\ y_c - p \\ z_c \end{bmatrix}. \tag{C.22}$$

$$\vec{V}_{12} = \begin{bmatrix} x_c + x_{12} \\ y_c + y_{12} \\ z_c + z_{12} \end{bmatrix} = \begin{bmatrix} x_c - q \\ y_c - p \\ z_c \end{bmatrix}. \tag{C.23}$$

More details about the icosahedron is available from [24].

C.5 Dodecahedron

The dodecahedron is a solid geometry composed of twenty vertices, thirty edges, and twelve pentagonal faces as shown in figure C.5 - create by Robert Webb and available from [20]. In order to calculate the vertex positions of the do-

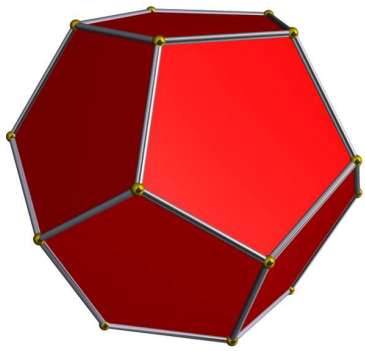


Figure C.5: The dodecahedron is one of the five Platonic solids with twenty vertices, thirty edges, and twelve faces.

decahedron, a number of useful constants and expressions are defined. In the equations that follow, two angles, denoted ϕ_1 and ϕ_2, having values $\phi_1 = 52.62$ and $\phi_2 = 10.81$, are used extensively. Another useful expression, which enters the position equations, is the radius of the circumscribed sphere, denoted R, which can be expressed as a function of the edge length a.

$$R = \frac{\sqrt{15}+\sqrt{3}}{4}a \tag{C.24}$$

In the following the positions of the vertices are calculated. This is done by dividing the vertices into four groups, where each group contains vertices with similar properties. Specific details about the geometric properties of the dodecahedron is available in [25].

Vertices V1 to V5: Let $\vec{V}_i$ be a three-dimensional vector describing the vertex positions of a dodecahedron, where $i = \{1, 2, ..., 5\}$. Then the positions of the first five vertices can be expressed as

$$\vec{V}_i = \begin{bmatrix} x_c + x_i \\ y_c + y_i \\ z_c + z_i \end{bmatrix} = \begin{bmatrix} x_c + R\cos(\theta_i)\cos(\phi_a) \\ y_c + R\sin(\theta_i)\cos(\phi_a) \\ z_c + R\sin(\phi_a) \end{bmatrix}, \tag{C.25}$$

where

$$\phi_a = \frac{\pi}{180}\phi_1 \tag{C.26}$$

and

$$\theta_i = \theta_{i-1} + \frac{72\pi}{180} \tag{C.27}$$

where $\theta_0 = 0$.

Vertices V6 to V10: For $i = \{6, 7, ..., 10\}$ the following vertex positions can be expressed as

$$\vec{V_i} = \begin{bmatrix} x_c + x_i \\ y_c + y_i \\ z_c + z_i \end{bmatrix} = \begin{bmatrix} x_c + R\cos(\theta_i)\cos(\phi_b) \\ y_c + R\sin(\theta)_i\cos(\phi_b) \\ y_c + R\sin(\phi_b) \end{bmatrix}, \tag{C.28}$$

where

$$\phi_b = \frac{\pi}{180}\phi_2 \tag{C.29}$$

and

$$\theta_i = \theta_{i-1} + \frac{72\pi}{180} \tag{C.30}$$

where $\theta_5 = 0$.

Vertices V11 to V15: For $i = \{11, 12, ..., 15\}$ the following vertex positions can be expressed as

$$\vec{V_i} = \begin{bmatrix} x_c + x_i \\ y_c + y_i \\ z_c + z_i \end{bmatrix} = \begin{bmatrix} x_c + R\cos(\theta_i)\cos(\phi_c) \\ y_c + R\sin(\theta_i)\cos(\phi_c) \\ z_c + R\sin(\phi_c) \end{bmatrix}, \tag{C.31}$$

where

$$\phi_c = -\frac{\pi}{180}\phi_2 \tag{C.32}$$

and

$$\theta_i = \theta_{i-1} + \frac{72\pi}{180} \tag{C.33}$$

where $\theta_{10} = \frac{72\pi}{360}$.

Vertices V16 to V20: For $i = \{16, 17, ..., 20\}$ the following vertex positions can be expressed as

$$\vec{V_i} = \begin{bmatrix} x_c + x_i \\ y_c + y_i \\ z_c + z_i \end{bmatrix} = \begin{bmatrix} x_c + R\cos(\theta_i)\cos(\phi_d) \\ y_c + R\sin(\theta_i)\cos(\phi_d) \\ z_c + R\sin(\phi_d) \end{bmatrix}, \tag{C.34}$$

where

$$\phi_d = -\frac{\pi}{180}\phi_1 \tag{C.35}$$

and

$$\theta_i = \theta_{i-1} + \frac{72\pi}{180} \tag{C.36}$$

where $\theta_{15} = \frac{72\pi}{360}$.

Online Resources

[1] Brian Horn and Bjørn Grønbæk. Investigation of elements for an automated test tool for virtual environments. Technical report, University of Southern Denmark - Mærsk Mc-Kinney Møller Institute, 2008.

[2] Confluence. Welcome to cruisecontrol.net. Website, 2008. http://confluence.public.thoughtworks.org/display/CCNET/Welcome+to+CruiseControl.

[3] Sourceforge. Nant : A .net build tool. Website, 2008. http://nant.sourceforge.net/.

[4] Alexei Berillo. Analysing nvidia g8x performance in modern games. Website, 2007. http://www.digit-life.com/articles2/video/g80_units2.html.

[5] Ding-Yun Chen, Xiao-Pei Tian, Yu-Te Shen, and Ming Ouhyoung. On visual similarity based 3d model retrieval. *Computer Graphics Forum (EUROGRAPHICS' 03)*, 22(3):223–232, 2003.

[6] Eric W. Weisstein. Archimedean solid. Website, 2009. http://mathworld.wolfram.com/ArchimedeanSolid.html.

[7] Eric W. Weisstein. Platonic solid. Website, 2009. http://mathworld.wolfram.com/PlatonicSolid.html.

[8] Edward H. Adelson. Shading effects can cause visual illusions. Website, 2008. http://tywkiwdbi.blogspot.com/2007/12/checkershadow-optical-illusion.html.

[9] Games from Within. Exploring the c++ unit testing framework jungle. Website, 2004. http://www.gamesfromwithin.com/articles/0412/000061.html.

[10] Sourceforge. Delta3d : A c++ game engine. Website, 2008. http://www.delta3d.org/.

[11] Sourceforge. Delta3d wiki. Website, 2008. http://delta3d.wiki.sourceforge.net/.

[12] Open Scene Graph SVN. Open scene graph svn trunk. Website. http://www.openscenegraph.org/svn/osg/OpenSceneGraph/trunk.

[13] Mattias Helsing. Open scene graph dependencies. Website. http://www.openscenegraph.org/projects/osg/wiki/Community/People/MattiasHelsing.

[14] Confluence. Configuring the server. Website, 2008. http://confluence.public.thoughtworks.org/display/CCNET/Configuring+the+Server.

[15] X.Y.Z. Building native c++ projects with nant. Website, 2006. http://seclib.blogspot.com/2005/05/building-native-c-projects-with-nant.html.

[16] Confluence. The server service application. Website, 2008. http://confluence.public.thoughtworks.org/display/CCNET/The+Server+Service+Application.

[17] Microsoft Developer Network. Msbuild reference. Website, 2008. http://msdn2.microsoft.com/en-us/library/0k6kkbsd(VS.80).aspx.

[18] Nick Parker. Adding visual studio .net 2003 support for nant. Website, 2006. http://developernotes.com/post/Adding-Visual-Studio-NET-2003-Support-for-NAnt.aspx.

[19] Jean-Paul S. Boodhoo. Nant starter series. Website, 2006. http://www.jpboodhoo.com/blog/NAntStarterSeries.aspx.

[20] Robert Webb. Images of platonic solids. Website, 2009. http://www.software3d.com/Stella.php.

[21] Eric W. Weisstein. Tetrahedron. Website, 2009. http://mathworld.wolfram.com/Tetrahedron.html.

[22] Eric W. Weisstein. Hexahedron. Website, 2009. http://mathworld.wolfram.com/Hexahedron.html.

[23] Eric W. Weisstein. Octahedron. Website, 2009. http://mathworld.wolfram.com/Octahedron.html.

[24] Eric W. Weisstein. Icosahedron. Website, 2009. http://mathworld.wolfram.com/Icosahedron.html.

[25] Eric W. Weisstein. Dodecahedron. Website, 2009. http://mathworld.wolfram.com/Dodecahedron.html.

Bibliography

[Alface et al., 2005] Alface, P. R., Craene, M. D., and Macq, B. (2005). Three-dimensional image quality measurement for the benchmarking of 3d watermarking schemes. *SPIE proceedings series*, 7:230–240.

[Avcıbaş et al., 2002] Avcıbaş, I., Sankur, B., and Sayood, K. (2002). Statistical evaluation of image quality measures. *Journal of Electronic Imaging*, 11:206–223.

[Bierbaum et al., 2003] Bierbaum, A., Hartling, P., and Cruz-Neira, C. (2003). Automated testing of virtual reality application interfaces. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 107–114, New York, NY, USA. ACM.

[Bosch et al.,] Bosch, J., Molin, P., Mattsson, M., and Bengtsson, P. Object-oriented frameworks- problems & experiences.

[Craig, 1989] Craig, J. J. (1989). *Introduction to Robotics: Mechanics and Control*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Darken, 2004] Darken, C. (2004). Visibility and concealment algorithms for 3d simulations. MOVES Institute - Naval Postgraduate School - California USA.

[Darken et al., 2007] Darken, C., Anderegg, B., and McDowell, P. (1-5 April 2007). Game ai in delta3d. *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 312–319.

[Eberly, 2001] Eberly, D. H. (2001). *3D Game Engine Design*. Morgan Kaufmann.

[Garland and Heckbert, 1997] Garland, M. and Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Gerrard, 1997] Gerrard, P. (1997). Testing gui applications. Presented at EuroSTAR 1997. http://www.gerrardconsulting.com/GUI/TestGui.html.

[Horn and Grønbæk, 2008] Horn, B. and Grønbæk, B. (2008). Investigation of elements for an automated test tool for virtual environments. Technical report, University of Southern Denmark - Mærsk Mc-Kinney Møller Institute.

[Ives et al., 1999] Ives, R., Eichel, P., and Magotra, N. (1999). A new sar image compression quality metric. *Circuits and Systems, 1999. 42nd Midwest Symposium on*, 2:1143–1146 vol. 2.

[Jain, 1989] Jain, A. K. (1989). *Fundamentals of digital image processing*. Prentice Hall Information and System Sciences Series, Englewood Cliffs: Prentice Hall, 1989.

[Josuttis, 1999] Josuttis, N. M. (1999). *The C++ Standard Library : A Tutorial and Reference*. Addison-Wesley Professional.

[Kenneth E. Hoff, 1997] Kenneth E. Hoff, I. (1997). Faster 3d game graphics by not drawing what is not seen. *Crossroads*, 3(4):20–23.

[Laga et al., 2004] Laga, H., Takahashi, H., and Nakajima, M. (2004). Geometry image matching for similarity estimation of 3d shapes. In *CGI '04: Proceedings of the Computer Graphics International*, pages 490–496, Washington, DC, USA. IEEE Computer Society.

[Lindstrom, 2000] Lindstrom, P. (2000). *Image-Driven Simplification*. PhD thesis, Georgia Institute of Technology.

[Lindstrom et al., 1996] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G. A. (1996). Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA. ACM.

[Lindstrom and Turk, 2000] Lindstrom, P. and Turk, G. (2000). Image-driven simplification. *ACM Trans. Graph.*, 19(3):204–241.

[Llopis, 2003] Llopis, N. (2003). *C++ For Game Programmers*. Charles River Media.

[Martz, 2007] Martz, P. (2007). *Open Scene Graph Quick Start Guide A Quick Introduction to the Cross-Platform Open source Scene Graph API*. Skew Matrix Software.

[McDowell et al., 2005] McDowell, P., Darken, R., Sullivan, J., and Johnson, E. (2005). Delta3d: A revolution in the methods of building training simulators. MOVES Institute - Naval Postgraduate School - California USA.

[McDowell et al., 2006] McDowell, P., Darken, R., Sullivan, J., and Johnson, E. (2006). Dela3d: A complete open source game and simulation engine for building military training systems. *JDMS*, 3(3):143–154. The Society for Modeling and Simulation International.

[Memon, 2002] Memon, A. M. (2002). Gui testing: Pitfalls and process. *Software Technologies*, pages 87–88.

[Memon and Soffa, 2003] Memon, A. M. and Soffa, M. L. (2003). Regression testing of guis. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA. ACM.

[Meyers, 2001] Meyers, S. (2001). *Effective STL 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley.

[MSDN, 2008] MSDN (2008). Regression testing. Website. http://msdn.microsoft.com/en-us/library/aa292167(VS.71).aspx.

[Onoma et al., 1998] Onoma, A. K., Tsai, W.-T., Poonawala, M., and Suganuma, H. (1998). Regression testing in an industrial environment. *Commun. ACM*, 41(5):81–86.

[Ramasubramanian et al., 1999] Ramasubramanian, M., Pattanaik, S. N., and Greenberg, D. P. (1999). A perceptually based physical error metric for realistic image synthesis. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 73–82, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Shade et al., 1998] Shade, J., Gortler, S., wei He, L., and Szeliski, R. (1998). Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, New York, NY, USA. ACM.

[Sibai, 2007] Sibai, F. (11-13 July 2007). 3d graphics performance scaling and workload decomposition and analysis. *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, pages 604–609.

[Slabaugh, 2008] Slabaugh, G. G. (2008). Computing euler angles from a rotation matrix. Technical report.

[Sonka et al., 2007] Sonka, M., Hlavac, V., and Boyle, R. (2007). *Image Processing, Analysis, and Machine Vision*. Thomson-Engineering.

[Sun and Li, 2005] Sun, M. and Li, C. (2005). A new complex sar image compression quality metric. *Geoscience and Remote Sensing Symposium, 2005. IGARSS '05. Proceedings. 2005 IEEE International*, 7:4697–4700.

[Toet and Lucassen, 2003] Toet, A. and Lucassen, M. P. (2003). A new universal colour image fidelity metric. *Elsevire Displays*, 7:197–207.

[Van Gurp and Bosch, 2000] Van Gurp, J. and Bosch, J. (2000). Design, implementation and evolution of object oriented frameworks: concepts & guidelines. In *the Second International Workshop on Component-Oriented Programming," Proceedings of the 2nd International Workshop on Component-Oriented Programming*, volume 1, pages 1–25. John Wiley & Sons, Ltd.

[van Rossum, 1999a] van Rossum, G. (1999a). *Python Library Reference*. Corporation for National Research Initiatives (CNRI), 1895 Preston White Drive - Reston Va 20191 USA. guido@CNRI.Reston.Va.US - guido@python.org.

[van Rossum, 1999b] van Rossum, G. (1999b). *Python Reference Manual*. Corporation for National Research Initiatives (CNRI), 1895 Preston White Drive - Reston Va 20191 USA. guido@CNRI.Reston.Va.US - guido@python.org.

[Vanmali et al., 2002] Vanmali, M., Last, M., and Kandel, A. (2002). Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1):45–62.

[Wahl, 1999] Wahl, N. J. (1999). An overview of regression testing. *SIGSOFT Softw. Eng. Notes*, 24(1):69–73.

[Xie and Memon, 2007] Xie, Q. and Memon, A. M. (2007). Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4.