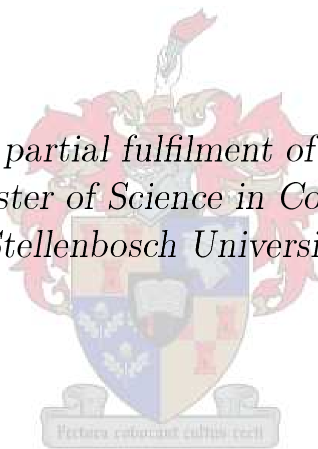
A 3D Virtual Environment Development Platform for ASD Therapy Tools

by

Morné Chamberlain



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at Stellenbosch University

Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. L van Zijl

December 2009

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: ....
ME Chamberlain

Date: ....

Copyright © 2009 Stellenbosch University
All rights reserved.

Abstract

A 3D Virtual Environment Development Platform for ASD Therapy Tools

ME Chamberlain

Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MSc (Ccomputer Science)

December 2009

The aim of this thesis is to develop a generic 3D virtual environment development platform for autism spectrum disorder (ASD) therapy tools. The potential of using computerised therapy tools for ASD therapy is well known. However, the development of such tools is expensive and time-consuming, and is language and culture specific. This work intends to alleviate these problems.

The design of the platform is based on known game engine designs, but adapted for the requirements of ASD therapy tools. It supports standard features such as 3D rendering, animation and audio output. Specific features, aimed at ASD therapy tools and educational games, included in our engine are: replays, data capturing, remote monitoring over a network and language localisation. We also implemented an input hardware abstraction layer to allow support for non-standard input peripherals in the future, without modifying existing game implementations. Furthermore, to separate the development of games and tools from the engine, we include wrapper libraries in our engine for Lua and Java.

We successfully developed our engine and implemented a number of prototype therapy tools and educational games. These implementations confirmed that the engine works as expected. Some of these programs are currently in use at a local primary school.

Uittreksel

'n 3D Virtuele Omgewing en Ontwikkelingsplatform vir OSV Terapiemiddels

("A 3D Virtual Environment Development Platform for ASD Therapy Tools")

ME Chamberlain

Departement Wiskundige Wetenskappe,
Afdeling Rekenaarwetenskap,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MSc (Rekenaarwetenskap)

Desember 2009

Die doel van hierdie tesis is om 'n 3D virtuele omgewing en ontwikkelingsplatform vir outistiese spektrum versteuring (OSV) terapiemiddels te ontwikkel. Die gebruik van rekenaargebaseerde terapiemiddels vir OSV terapie is bekend. Om sulke terapiemiddels te ontwikkel is egter duur, tydrowend en is dikwels gerig op spesifieke taal- en kultuurgroepe. Hierdie werk het dit ten doel om hierdie probleme te bowe te kom.

Die ontwerp van die platform is gebaseer op die ontwerp van bekende videospeletjie-enjins, maar is aangepas vir die benodigdhede van OSV terapiemiddels. Dit ondersteun standaard funksionaliteit soos 3D uitbeelding, animasie en klank. Ons platform sluit in spesifieke funksionaliteit, wat gerig is op OSV terapiemiddels en opvoedkundige speletjies, naamlik: kykweer, datavaslegging, afstandswaarneming oor 'n netwerk en taal-lokalisering. Verder is 'n abstrakte koppelvlak vir toevoerapparatuur ontwikkel, wat dit moontlik maak om in die toekoms nie-standaard toevoerapparatuur te ondersteun, sonder om bestaande speletjies se implementasies aan te pas. Verder, om die ontwikkeling van speletjies en terapiemiddels te skei van die enjin, is koppelvlakke ontwikkel wat dit moontlik maak om die enjin in Lua en Java te gebruik.

Ons enjin is suksesvol ontwikkel en 'n aantal prototipe terapiemiddels en opvoedkundige speletjies is daarmee ontwikkel. Hierdie prototipes het bevestig dat die enjin werk soos ons verwag. Sekere van hierdie programme is tans in gebruik by 'n plaaslike laerskool.

Acknowledgements

I would like to thank my supervisor, Prof. Lynette van Zijl, for her mentorship, motivation and patience throughout this thesis. I would also like to thank my family and friends for their continuous support.

The financial assistance of the National Research Foundation (NRF) and the Wilhelm Frank Bursary is hereby acknowledged. Opinions expressed and conclusions arrived at in this thesis, are those of the author and are not necessarily to be attributed to the NRF or the Wilhelm Frank Bursary.

Contents

Declaration i

Abstract ii

Uittreksel iii

Acknowledgements v

Contents vi

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Thesis Outline 3

2 Literature Survey 4

2.1 Existing VEs for ASD Therapy 4

2.2 Standard Engines 5

2.3 On-line Social Networking Virtual Environments 16

3 The Design of the Three-Dimensional Virtual Environment Platform 19

3.1 Introduction 19

3.2 The Task Management Interface 25

3.3 Managers 36

3.4 The Plug-in API 37

3.5 Collision Detection and Physics 38

3.6 The Game Object Hierarchy 40

3.7 Management of GameObjects 46

3.8 Rendering 48

3.9 Input Handling 49

3.10 Audio 52

3.11 Graphical User Interfaces 54
3.12 Data Capturing 56
3.13 Serialisation 58
3.14 Replays 59
3.15 The Networking API 64
3.16 Scripting 68

4 Implementation **70**
4.1 Multi-Threading and Thread Safety 70
4.2 The Timer Task and Message Passing 71
4.3 The Input System 74
4.4 Audio Plug-in 76
4.5 SQLite Data Capturing 76
4.6 Network 78
4.7 Scripting 80
4.8 Localisation 83
4.9 Content Creation and Loading 85

5 Educational Game and Therapy Tool Implementations **86**
5.1 I Am Special 86
5.2 Journey to the Moon 88
5.3 Social Skills 97

6 Testing **100**
6.1 Overview 100
6.2 Functional Tests 101
6.3 Coverage 104

7 Results and Conclusion **106**

Appendices **111**

A Design Patterns **112**
A.1 Singletons 112
A.2 Abstract Factory 113
A.3 Functor 113

B Educational Game Questionnaire **114**

Bibliography **119**

List of Figures

2.1 Design overview of the Open Game Engine, taken from [59]. 7
2.2 Object component system overview in the Open Game Engine, based on the object system diagram from [60]. 8
2.3 Design diagram of the Enginuity Engine, taken from [22]. 9

3.1 The Task class hierarchy. 28
3.2 Class A sends a message to class B via (a) asynchronous and (b) synchronous message passing. 36
(a) Class A sends a message to Class B asynchronously. 36
(b) Class A sends a message to Class B synchronously. 36
3.3 The hierarchy of classes that represent the various types of objects in the virtual world. 43
3.4 An overview of the hardware abstraction layer. Third-party libraries manage input hardware and communicate events and state changes to the input handling layer in the engine. This layer then creates InputMessages that encapsulate these input events. 49
3.5 The hierarchy of InputDevice classes. 50
3.6 Handling an input event. 52
3.7 An overview of the design of the audio framework. 53
3.8 An overview of the design of the GUI system in the engine. 55
3.9 An entity, as a table, with its entries. The underlined columns constitute the key for an entry. 57
3.10 An overview of the design of the data capturing framework. 58
3.11 The design of the replay models for recording and playback. 62
(a) The design of the replay model for recording a replay. 62
(b) The design of the replay model for playing a replay. 62
3.12 An example of the transmission of an InputMessage with the peer-to-peer network model. 66

5.1 A computerised worksheet from *I Am Special*. 87
5.2 Before the ship is released from its dock. 90

5.3 Before the exterior door is opened. 90
5.4 During ship take-off. 90
5.5 The start of the second scene. 91
5.6 Dodging an asteroid. 91
5.7 Approaching the Moon. 91
5.8 A portion of a FSM that could represent the game logic of the *Journey to the Moon* game. 92
5.9 The restaurant scenario in the social skills application. 97

List of Tables

6.1 Nine testing programs, P1–P9, with the engine features that are tested by these programs. 101
6.2 Function coverage percentages for various systems in the engine. 104

Chapter 1

Introduction

Autism spectrum disorders (ASDs) currently affect a significant percentage of the world-wide population [9, 13]. The potential of computerised therapy tools, in particular the use of 3D virtual environments, for ASDs is well known [14, 39, 43, 44, 50, 55], but the development of such tools is particularly time-consuming. The problem is further exacerbated by the fact that many ASD therapy methods have language and cultural dependencies, so that European and American implementations cannot simply be re-used in any South African context. Many previously developed computerised tools suffer from two common problems: the sources are not freely available and/or the implementation cannot easily be adapted to a South African context. This thesis investigates and discusses the creation of an open-source platform, that aims to alleviate the above-mentioned problems in the implementation of ASD therapy tools and general educational games.

This thesis forms part of a larger project on computerised assistive technologies for individuals with ASD, namely the ASD Assist project. In this thesis we investigate the creation of a development platform for 3D virtual environments for use in ASD therapy. This platform will in future be utilised by other sub-projects of the ASD Assist project, such as an avatar with the behavioural aspects of an individual with Asperger's syndrome or social skills therapy tools and educational games aimed at individuals with ASD.

Therefore, we embarked upon the design and implementation of a generic 3D virtual environment platform with the following goals. The platform must:

- be able to manage and render 3D virtual environments,
- be adaptable and extendable for specific needs,
- have longevity: it should remain useful in the future,
- be well documented,

- have a generic input management system,
- have a customisable graphical user interface (GUI) system,
- include a replay system,
- have a data capturing system,
- provide a remote monitoring facility where therapists can monitor a session over a network,
- provide multi-lingual support in games and therapy tools through a language localisation system, and
- provide a framework upon which tools can be built to aid non-programmers in constructing educational games and ASD therapy tools.

The first goal listed above states that our platform has to be able to manage and render 3D virtual environments. This includes managing input from a player, and managing output, such as graphics and audio, from the 3D virtual environment. Since libraries for 3D virtual environments are typically large we shall rely on a number of excellent open-source libraries to provide some of the functionality we require, instead of implementing every aspect ourselves. Some of the open-source libraries we consider are the 3D game and/or graphics engines Panda3D [61], Crystal Space [58], the Enginuity Engine [22] and OGRE [53].

Our platform has to have longevity. Our platform forms part of a larger project and must remain useful in the future. We aim to add basic support for multi-threading in our platform since multi-core CPUs are becoming more and more standard. This could help to keep the engine relevant in the future. Furthermore, we strive to create a platform with above average performance, aimed at low entry-level hardware requirements. In the South African context, not all schools and therapists would necessarily be able to afford expensive hardware.

Our platform must support a generic input system to allow for future extensions that could add support for diverse input devices, such as speech recognition for example. Individuals with ASD often suffer from associated motor difficulties [26, 27] that make it difficult for them to use standard input peripherals, such as a keyboard and mouse.

Applications and games that are aimed at individuals with ASD require carefully designed GUIs. Certain elements, such as flashing icons, can be distracting and could make such GUIs unusable for individuals with ASD [36]. It is therefore important that our platform supports a customisable GUI system.

Therapists [23] indicated that it would be invaluable to capture data about certain events or actions that a user takes while engaged in a therapy tool or educational game. Therefore, we include two methods of data capturing in the goals of this project. The first method is a simple replay system, similar to what is common in sports games or strategy games. The second method provides a framework whereby more precise data can be captured. The nature of the data captured and the events that are captured are defined by the implementers of the educational games or therapy tools, for each game or tool. Lastly we also include a network-based remote monitoring system as a goal of this project. This allows therapists to monitor a user engaged in a therapy tool in real-time, from another computer on a network.

One of our goals is to provide a framework upon which tools can be built that non-programmers can utilise to build 3D virtual environments that utilise our platform. We propose to use a scripting language interface to our platform for such a framework. The goal is that such a scripting language interface could be used to implement game and therapy tools that utilise our platform. These scripts would then also be the output of envisioned visual tools that could be used by non-programmers to construct educational games and therapy tools.

1.1 Thesis Outline

We now discuss the layout of the remainder of this thesis. In Chapter 2 we discuss existing 3D virtual environment platforms as well as research on the use of computerised ASD therapy tools. In Chapter 3 there is an in-depth discussion around the design of our 3D virtual environment platform for educational games and ASD therapy tools. Chapter 4 discusses a number of issues we encountered during the implementation of our design along with the solutions to these issues. Chapter 5 discusses the creation of a computerised version of a portion of the popular *I Am Special* [66] workbook used for ASD therapy. The chapter also discusses the creation of one general educational game and one social skills therapy tool aimed at individuals with Asperger's Syndrome (AS). All of the games and tools discussed in Chapter 5 were created using our platform. In Chapter 6 we discuss the methods we utilised to test the engine during its implementation. Chapter 7 discusses the ultimate strengths and weaknesses of our 3D virtual environment platform implementation and how well it satisfies the initial goals we defined for the project. The chapter also states our conclusions about the project and possible future work.

In the next chapter we survey existing engine implementations as well as the current research into computerised ASD therapy tools.

Chapter 2

Literature Survey

In this chapter, we survey a number of virtual environment engines in order to determine the usefulness of these existing engines for our project. We also consider a number of common design elements among these engines. Furthermore, we discuss studies relating to the use of virtual environments in ASD therapy tools. In particular, we cover the work describing trials of an existing software package that aspires to teach social skills to children with ASDs.

In the next section we survey research that have been conducted into using VEs as ASD therapy tools.

2.1 Existing VEs for ASD Therapy

We begin this section with a brief discussion on previous research about the use of virtual environments for learning, and specifically the use of virtual environments for ASD therapy. Cobb *et al.* [15] evaluated the use of virtual environments as learning environments for individuals with and without special needs. The use of virtual environments in this regard was found to be promising, but certain individuals required more assistance with the use of the VE input devices than others. This indicates that careful design of VEs is necessary in order to support as many input device types as possible so that such VEs are usable by as wide an audience as possible. Strickland [55] researched the use of virtual reality to treat individuals with autism and found its use promising. Moore *et al.* [39] recognised the potential benefit of computerised learning for individuals with autism and presented a framework for further research in the field. In 2005 Moore *et al.* [14] researched the use of collaborative virtual learning environments focussed on individuals with autism. In collaborative virtual environments (CVEs) individuals join the environment from their own computers and each user is represented by an avatar. Moore *et al.*

states that this allows individuals with autism to interact in the VE without the same fear and anxiety they might experience with face-to-face communication. Furthermore, users in this CVE can express their feelings by selecting emotional states for their avatar. In 2002 Parsons *et al.* [43] investigated the potential for using virtual environments to teach social skills and social understanding to individuals with ASDs. They found the use of VEs promising since it offers safety and enables individuals to repeatedly practise tasks in a consistent manner. In 2007 Parsons *et al.* [44] conducted a study where they attempted to use virtual environments to teach social skills and social understanding to six individuals with ASDs. The study tested whether participants showed improvement in their social skills after some exposure to a VE that was constructed to teach social skills. The results of the study showed that careful design is necessary, but that VEs do indeed have the potential to improve the social understanding of individuals with ASDs. In 2008 Schmidt *et al.* [50] investigated the use of virtual environments in teaching social skills to individuals with ASD.

Each of these individual groups, conducting research into using VEs for ASD therapy, have to develop specific software for creating their particular VEs. Our aim is to provide a general VE platform that could be used by these research groups. Each group can then focus on building their particular VEs, using our platform, instead of spending time to find or build engines for VE creation and management.

In the following sections, we discuss existing engines and the related literature. We broadly categorise the survey by the type of the engine or its intended usage. For each engine, we point out the salient design and usage features as it relates to our project. We begin by surveying some standard computer game engines.

2.2 Standard Engines

In this section we survey some existing game engines. We discuss the suitability of each engine based on how well the engine satisfies the goals set out for our platform in Chapter 1 and the potential time it would take to adapt the engine to our needs.

A comprehensive and representative survey of existing game engines is difficult, as the source code and information for commercial engines are not freely available. Where developers do release source code, it typically concerns older projects, and techniques are usually outdated and based on older technologies. One such example is the Quake 3 engine, where the source code was released by Id Software in 2005.

2.2.1 The Quake 3 Engine

We analysed the source code of the Quake 3 engine [30] directly, as there is no reference documentation or design documentation available for the engine. While it was an excellent engine when it was released, there is a number of troubling factors that complicate its use in our research today:

- it does not use a scripting language to the extent that we require,
- it relies on older file formats making it difficult to use with some of the latest third-party modelling applications, and
- it would take a prohibitive amount of time to study and modify the engine to suit our purposes.

We therefore decided not to use the Quake 3 engine in our research. As the sources for other commercial game engines were not obtainable, we focused our attention on open-source game engines. This lead us to the engine discussed in the next section, namely, the Open Game Engine.

2.2.2 The Open Game Engine

The Open Game Engine (OGE) [59] is an open-source community project that strives to create a powerful 3D computer game engine. It has several features that we require, such as high-level scripting language support. In particular, OGE uses Squirrel [19] as its scripting language. Squirrel is a fairly new scripting language that was designed with game development in mind. OGE makes extensive use of proven open-source projects as third-party libraries. The list of libraries include: Object-oriented Graphics Rendering Engine (OGRE) [53], Open Input System (OIS) [12] and the Squirrel scripting language.

OGE follows a client-server design, as illustrated in Figure 2.1. In this high-level overview, every system in the engine is managed by a manager class as depicted in the diagram. The base implementation of OGE provides managers for graphics, input, artificial intelligence, physics, networking, audio, a GUI, scripting and finally for the game implementation itself. These managers typically rely on third party libraries for their primary functionality. Every manager runs in its own thread and the arrows in the diagram indicate which manager initiates communication with another. There is an *Object Manager* instance that manages the creation and destruction of the individual manager classes.

Figure 2.1 shows that there is a clear division visible between a client and a server implementation. However, the design is intended to be flexible and equally effective for local only single player games or networked multi-player games.

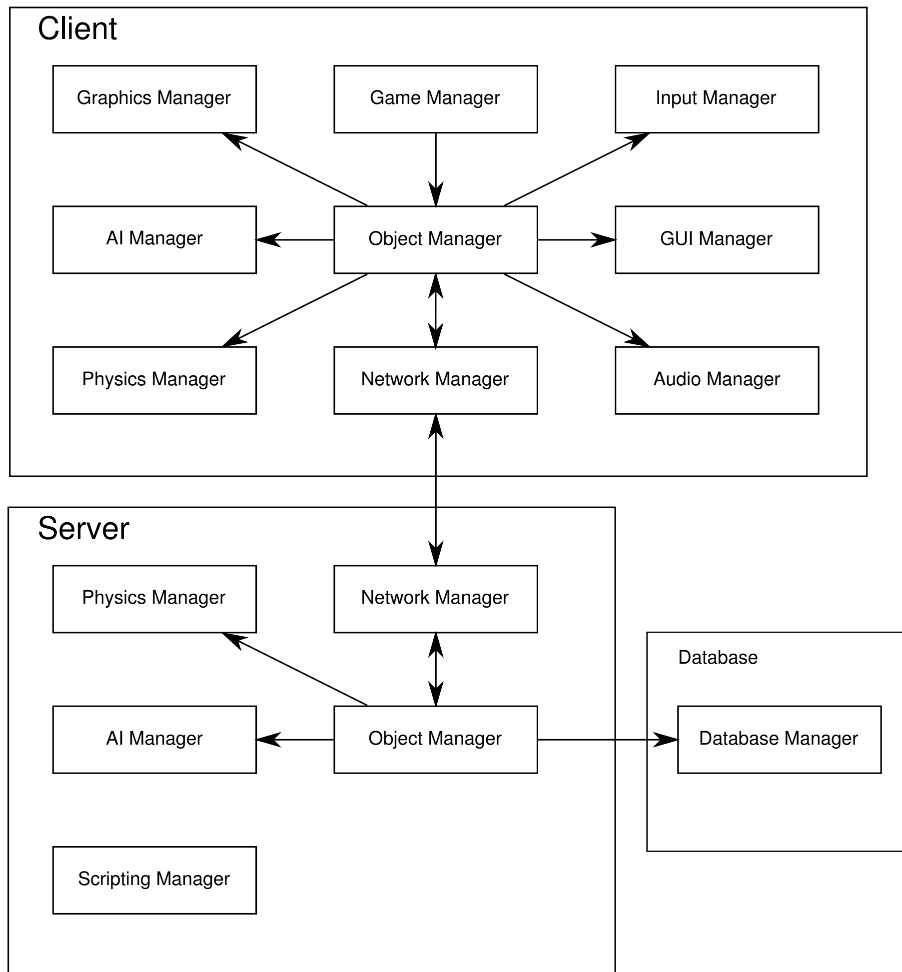


Figure 2.1: Design overview of the Open Game Engine, taken from [59].

The design of OGE also includes a database module. This allows a server implementation to store information that can be shared by multiple clients. It could also be useful for storing user profiles and for implementations that require users to log in before playing.

Another design feature of OGE is a component-based game object system. A game object in OGE can be extended by adding classes that implement a component interface to a local list of components for every object. This allows users of the engine to easily create customised game objects by using the scripting system of the engine. Figure 2.2 highlights that objects have a list of components and that various component implementations can be added to an object. Some examples of component implementations are objects that implement functionality for a physics simulation on the object or a so-called "mind" component that provides an object with artificial intelligence features. This component system allows for the extension of game objects dynamically and easily, without the need for hard-coding classes in the engine core.

The use of OGE for our research initially seemed promising. However, there are some factors that make its use restrictive. One such factor, and indeed the most important

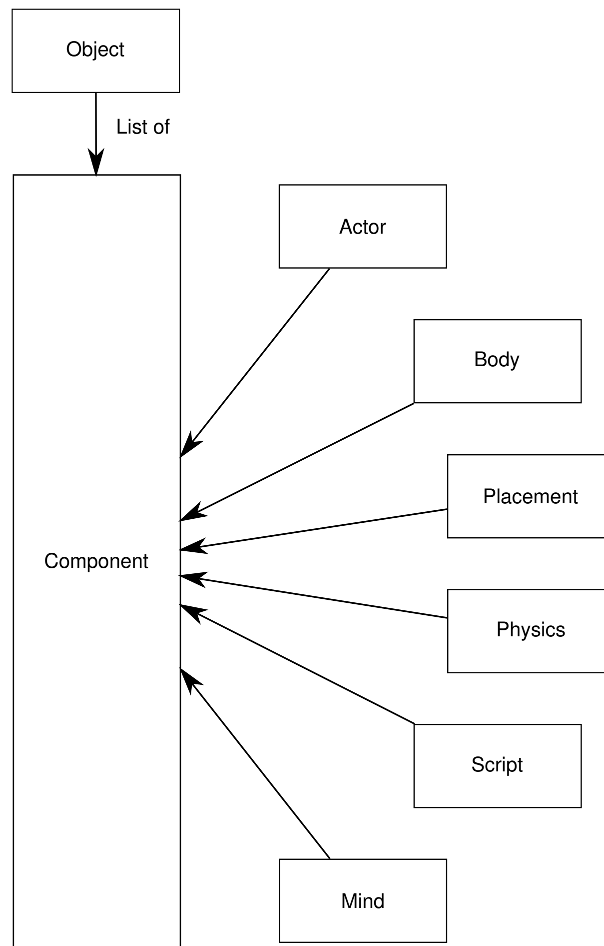


Figure 2.2: Object component system overview in the Open Game Engine, based on the object system diagram from [60].

problem, is that at the time of writing the engine does not function properly and is undergoing a complete redesign. Furthermore, while the engine follows a strong object-oriented design, some aspects are not generic enough to comply with all of our goals. For example, adding support for non-standard input devices, such as touch screens or speech recognition, is difficult. Therefore, the main reason why we decided not to use OGE was that we did not want to rely on an unproven system.

In the next section we discuss the Enginuity Engine.

2.2.3 The Enginuity Engine

The Enginuity Engine (EE) [22] was created by Richard Fine and the design and development of the engine is well-documented [22]. There is a C++ implementation of the EE available.

The design of the EE (taken from [22]) is outlined in Figure 2.3. The design is focused

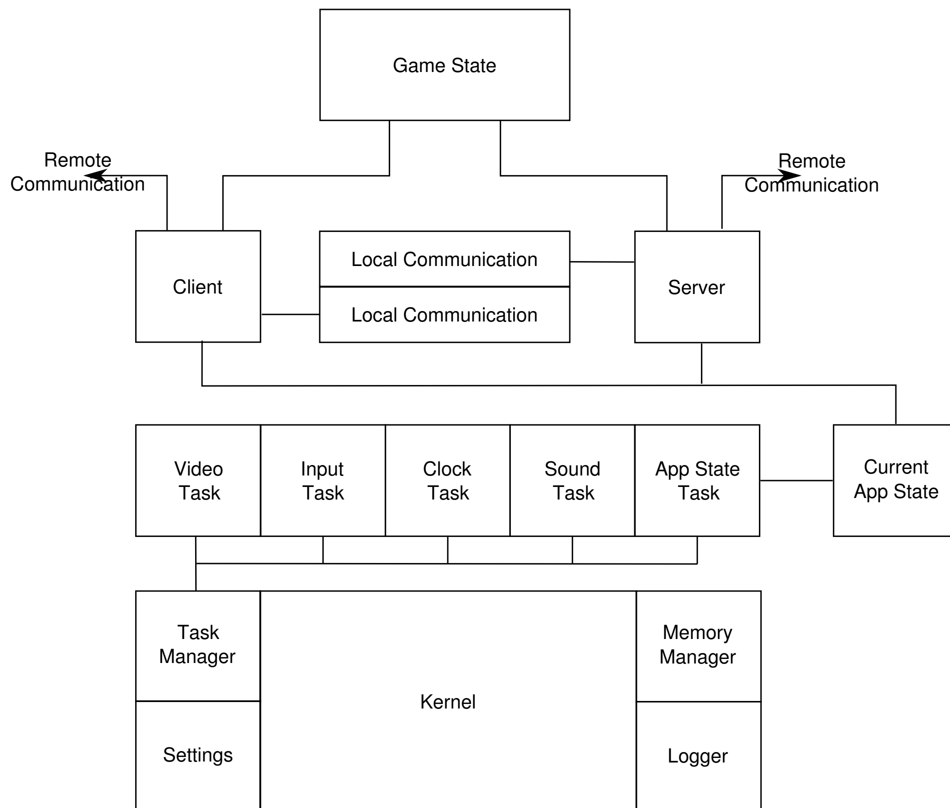


Figure 2.3: Design diagram of the Enginuity Engine, taken from [22].

around a number of tasks that provide services to the engine as a whole. The core of the engine is called the kernel. The kernel manages the creation and execution of tasks through the task manager. A list of common tasks is given in Figure 2.3. Among these is the so-called Clock Task. This task is responsible for controlling the timing mechanisms used in the engine. The kernel executes the tasks sequentially. The order of task execution is determined by the priority of each task. The Clock Task will have the highest priority and will always be updated first by the kernel. This design also implies that any other task with temporal dependencies would need to call a method from the Clock Task directly to determine the amount of time that has passed since a previous execution[1]. This direct communication and execution model favours a single-threaded engine design.

Another important design feature of the EE is that of memory management. Objects that are to be managed by the memory manager share a common ancestor in their inheritance hierarchies, namely, a reference counter. Memory management is done by reference counting, where objects that are left with no references in the engine are destroyed by the memory manager at certain intervals. This type of memory management is extremely important in game engines, as large numbers of objects are frequently created and discarded.

[1]Certain tasks require the time between executions, for example a physics task that solves a mathematical model in discrete time steps. This is discussed more in Chapter 3.

The EE design is based on a client-server model. A client and server both have a kernel and tasks. Generally, servers and clients will not have the same tasks exactly. For example, a dedicated server will not have a task for rendering graphics. The server and the clients both maintain the state of the game. Servers will periodically communicate the game state to all the connected clients, while the clients will send periodic updates of local changes to the server.

Furthermore, the design of the EE advances a separation between the engine and the application. Applications implement classes that extend existing engine classes (such as the game state from Figure 2.3). Applications then implement so-called factories to create these customised objects during run-time. This allows a high level of re-usability of the engine.

The EE has an extendable design. However, it is not a fully featured engine since only the engine core is available. Full support for graphics rendering, audio and input management is not available. It is also not a continuously developing open-source engine. Therefore, we cannot use it directly as a component in our platform. We can, if necessary, consider using some of the concepts introduced in its documentation.

In the following section we discuss a commercial quality open-source 3D game engine that was originally designed by *Walt Disney Entertainment*, namely the Panda3D engine.

2.2.4 Panda3D

Panda3D [61] is a 3D game engine originally developed by *Walt Disney Entertainment* but is currently maintained by the Entertainment Technology Center at Carnegie Mellon University. Panda3D is an extensive 3D game engine that is being used in commercial projects, such as *Pirates of the Caribbean On-line*.

The Panda3D engine consists of a number of data structures and systems common to modern 3D computer game engines. Any modern 3D rendering system uses a scene graph [51] implementation to manage the 3D objects that it renders. A scene graph is generally implemented with a data structure such as an octree [17]. An octree allows for the storage of numerous objects and has the advantage that it allows efficient searches for objects, based on positional information. The scene graph implementation is important for many of the other systems in the engine as it provides an efficient way to determine what objects are located in a certain area in the world and it allows one to determine whether certain objects are within close proximity of one another. In order to simulate basic Newtonian physics in game worlds, Panda3D also includes a physics system. The physics simulation included with Panda3D can be disabled if it is not required. Along with the physics system there is also a collision detection implementation.

Panda3D also has an audio subsystem. Thus it is a simple matter to include sound effects and music in a project implemented with Panda3D.

It is also helpful to have a powerful graphical user interface (GUI) implementation in a game engine. In our research we are concerned with creating intuitive user interfaces that are also aimed at individuals with ASD. Such individuals tend to have reduced attention span or can be easily confused by ill conceived user interfaces [36]. Panda3D includes a GUI system that is called DirectGUI. It provides good functionality for creating user interfaces with the engine.

As with the OGE and the EE, Panda3D also includes a task management system. Users of the engine can create tasks and schedule them for execution at certain intervals and with certain priorities. Panda3D also includes an event system where user-specified functions are called on certain engine events, such as a key press or mouse movement.

The Panda3D engine is programmed in C++, but the engine has a full Python interface that is automatically generated from the C++ interface. Thus all of the functionality available through the public C++ application programming interface (API) is available through the Python API as well. Such a scripting API has become standard in many game engines [17]. It allows for a simpler, high-level interface that can be used by individuals that lack the complex programming expertise required to work with the lower-level C++ API and source code. In the modern game development industry game designers frequently build most of the game-logic using the scripting API. Also, higher-level scripting languages are generally not compiled before execution, as is the case with traditional languages, but interpreted when executed. Thus it is faster to make and test a modification than with traditional languages, that would need to be recompiled with every new set of changes. Many engines also allow access to a scripting system during run-time, allowing testers to test and modify certain parameters during run-time. Thus modifications to a game, or in our case a therapy tool, can be made without recompiling the core engine but just by modifying the scripts.

Panda3D does not have built-in support for the data capturing functionality that we require. It does have a recording system that could act as the basis for a replay system. It also supports basic networking, such as managing sockets, but not complete management of a networked game session.

One of the problems with Panda3D is its sheer magnitude. The API specification is large and complex. There is a reasonably complete user manual available, but the API reference documentation is often incomplete or ambiguous. There is little public documentation available about the core design of the engine. An understanding of the engine design is crucial if we aim to extend engine functionality. This is problematic, as one of our goals is to create an extendable virtual environment engine.

Panda3D is a good example of an engine with some of the functionality that we require in our virtual environment engine. The Python interface is also a good example of the level of scripting support that we would require from our engine. In order to achieve the goals set out for our research, our virtual environment engine would be similar to Panda3D, with the following significant differences:

- Panda3D is a large and complicated system that is difficult to extend and we need an engine that is smaller and more focused on ease of use and extendability,
- the engine API must be well documented so that future extensions are simplified,
- the design and the design philosophy of the engine must be well documented, and
- access to scripting languages other than Python would be advantageous.

In the following section we discuss a mature open-source 3D rendering engine called Crystal Space.

2.2.5 Crystal Space

Crystal Space [58] is a cross-platform open-source software development kit for realtime 3D graphics. Crystal Space has a particular focus on game development. We briefly discuss the key features of Crystal Space.

Crystal Space is a mature library that has been in development for a number of years. The primary functionality of Crystal Space is that of a graphics engine. It has a scene graph API that is used to control the transformations of renderable entities in the game world. Extra functionality is added to the engine via plug-ins. Some notable plug-ins that have been created are a plug-in for adding sound and music, a physics plug-in that uses the Open Dynamics Engine (ODE) for physics and collision detection, and a GUI plug-in for Crazy Eddie's GUI System (CEGUI) [63]. The engine also has various wrapper libraries so that it can be utilised from scripting or higher level languages such as Java, Perl and Python. Python is the best supported of these wrappers and is used as the primary scripting language for the engine.

A further extension of the Crystal Space engine is the so-called Cell Entity Layer (CEL). CEL adds specific support, aimed at game development, to the engine. CEL provides entity management, scripting, and utilities to package a game for deployment.

Content creation for the Crystal Space engine is primarily done with Blender [57], since a full exporter from Blender to Crystal Space is available. Crystal Space also supports the *3ds* format used by 3D Studio Max [8].

The Crystal Space engine does have some limitations. The engine has a rendering plug-in API and currently has a software renderer plug-in and an OpenGL based hardware renderer plug-in. Some users might prefer to use Microsoft®'s DirectX, especially considering some of the recent advanced features added in DirectX 10 and 10.1. There is currently no DirectX based renderer available for Crystal Space. At the time of writing the Crystal Space engine also is not multi-threaded and does not fully utilise multi-processor architectures. A 2008 Google Summer of Code project aimed to add limited multi-threaded support for background resource loading to the engine [28]. Crystal Space also does not have built-in data capturing or replay recording functionality. There is a plug-in available for basic networking support, but not on the level that we require.

The Crystal Space engine has the following features that are useful for our research:

- mature 3D rendering library,
- support for scripting languages,
- extendable with plug-ins,
- audio support via a plug-in,
- physics and collision detection support via a plug-in, and
- a content exporter available for Blender.

The following features are lacking or problematic:

- the engine only supports OpenGL for hardware based rendering,
- does not have built-in support for data capturing,
- no built-in replay support, and
- the engine is not multi-threaded.

In the following section we discuss the Blender 3D modelling and animation package along with its integrated game engine.

2.2.6 The Blender Game Engine

Blender [57] is an open-source 3D modelling and animation package. Artists use Blender to create animated movies, visual effects and game content. The Blender project includes a game engine that is integrated into the primary modelling and animation software. It is

possible to create a scene or level, set up game logic, and then immediately test the scene or level from the primary modelling application instead of running an external executable.

Games that use the Blender engine are primarily implemented from within the modelling application or by using the Python API of the engine. Implementing logic from within the modelling application is done in a semi-visual manner, using a chain of three constructs called sensors, controllers and actuators. A sensor can be set on any object. The sensor waits for some event, such as input from the mouse and keyboard, and then passes this information along the chain to the controller. The controller on the object then determines what action should be taken; for example, should a cube be moved forward or rotated? The actual act of moving or rotating an object is done via an actuator. An object can have many sensors, controllers and actuators linked to one another under different conditions. The Blender game engine also supports physics and collision detection via the Bullet [4] library. The engine currently only supports the *wav* sound format.

A manual and the documentation for the Python API of the game engine is available. Some of the documentation is incomplete and appears to be out of date. The engine currently does not have an in-game graphical user interface (GUI) system, which complicates the creation of an in-game menu system. The engine also does not have built-in functionality for data capturing or replay recording. The Blender engine is not multi-threaded and does not fully utilise multi-processor architectures. We also could find no reference to support for networking in the engine.

Thus the Blender game engine provides us with the following useful features:

- logic scripting via the primary modelling application,
- quick testing of levels directly from the modelling application,
- Python API for more advanced logic scripting,
- physics and collision detection via Bullet, and
- audio support.

The Blender game engine lacks the following features:

- no built-in GUI support,
- only *wav* sounds are supported,
- no built-in data capturing support,
- no existing replay support,

- no known networking support,
- no multi-threading, and
- some documentation is incomplete or out of date.

In the following section we discuss the OGRE graphics rendering engine. Although not a full game engine, it still has numerous useful features.

2.2.7 OGRE

In this section we discuss the Object-oriented Graphics Rendering engine [32, 53]. As its name suggests, OGRE is a rendering engine and not a complete game engine. It can render a 3D scene, but it does not have built-in support for audio, physics and scripting for example. However, OGRE is designed to be extended by plug-ins and the highly active community of OGRE users and developers have developed numerous plug-ins and add-ons. Plug-ins generally extend existing functionality in OGRE by adding extended versions of existing classes. Add-ons typically add entirely new functionality. By using OGRE in conjunction with these plug-ins and add-ons, one can get similar functionality to that of most open-source game engines, with the exception of task management, communication and a specialised entity management system.

OGRE uses a scene graph data structure to store transformation based entities. Scene graph implementations can be switched by using plug-ins: this allows one to select an implementation that is well suited for the type of scene under construction. Examples include an octree based implementation, a terrain (height map) implementation, or a paged scene manager for large expansive scenes where portions of the scene are loaded to or unloaded from memory as the camera moves around the scene. OGRE supports key-framed animations and meshes with skeletal animations. It uses a generic rendering system with existing implementations for OpenGL, Microsoft® Direct3D 9 and Microsoft® Direct3D 10.

OGRE was designed with a simple API. It is well documented with numerous tutorials and an active developer and user community. Some plug-ins and add-ons created by the community include:

OgreOggSound: A plug-in that adds support for positional sound using OpenAL.

MyGUI: An add-on that manages graphical user interfaces rendered by OGRE. It includes a layout editor and themes defined in XML files.

CEGUI: Another GUI add-on.

OgreODE: An add-on that adds support for physics simulations to OGRE using ODE.

OgreNewt: Another physics simulation add-on using Newton.

OgreBullet: A physics add-on that uses Bullet.

NxOgre: A physics add-on that uses Nvidia PhysX.

OgreDotScene: A plug-in to load scene layout files in the Dotscene format, generally exported from Blender.

OgreCollada: A plug-in to load scenes and meshes in the Collada format.

There are also OGRE wrapper libraries available for certain scripting languages, such as pyOGRE for Python. OGRE has its own file format for meshes and skeletons, but converters and exporters are available for Blender and 3D Studio Max. Levels or scenes can be created in Blender or 3D Studio Max and exported to a format for which there is a plug-in for OGRE (for example, Dotscene and Collada).

Although OGRE, along with its plug-ins and add-ons, has a broad functionality, it is still not a complete game engine: it lacks a task management and communication system. Also, as with the other engines we discussed, OGRE does not have a data capturing system, replay system or networking plug-in. However, this does not mean it is a poor choice in our situation. Using OGRE and implementing a task management and communication system gives us more control and allows us to create a smaller, more focussed API than those of other engines. The multitude of plug-ins and add-ons that are available for OGRE gives us the luxury of choice in many situations. For example, there are four physics add-ons available. This means that we have the opportunity to evaluate the add-ons based on features and performance and choose one that suits our needs. OGRE is also one of the few engines with add-ons for robust GUI libraries. Both the MyGUI and CEGUI add-ons have so-called layout editors, allowing one to easily create GUI layouts with a what you see is what you get (WYSIWYG) editor.

Having discussed various game engines in some detail, we now consider whether on-line social networking virtual environments can be used to successfully implement ASD therapy tools and educational games.

2.3 On-line Social Networking Virtual Environments

Second Life [37] is one of the best-known on-line social networking virtual environments. We use it as an example to evaluate the possibility of using an on-line social VE instead of a standard game engine as the core of our intended platform.

Second Life was developed by Linden Labs. Users select and configure personal avatars to represent themselves in the environment. The Second Life world is divided into a public world and private regions. Any user can access the public world and interact with other users in the environment. Interactions vary but are generally based on sending text messages or by observing the behaviour of the avatar of a user.

Second Life has taken the concept of a virtual society to an extreme level, allowing users to buy and sell objects in the virtual world using Linden dollars, a currency managed by Linden Labs and created specifically for Second Life. Linden dollars can be exchanged for real world currency, such as United States dollars. This has lead to users turning profits from selling items or services in Second Life. Second Life also allows the user to buy land in the public world, as well as to purchase or rent private regions. Private regions are generally islands. Users can terraform these islands to create personalised designs, or they can choose from some existing designs. These private islands can then be used as locations to where friends or colleagues can be invited. A popular use is the creation of virtual training and conference centres.

Second Life has a “create anything” philosophy where certain tools are provided to allow the user to create anything they can imagine in the virtual world. Among these tools is the Linden Scripting Language (LSL). A user can use the LSL to easily implement behaviours for any objects that she creates. An example mentioned on the Second Life website is to create a butterfly and then scripting it to follow the avatar of the user around the world. Other tools are also provided, such as an editor to aid in the creation of custom objects in the world.

We were interested in Second Life for our research, as it provides a 3D virtual environment platform that can be used to create custom content. The LSL would make it possible to create custom scenarios for a user to complete in the virtual world. Furthermore, the sources of the Second Life client application are available. Hence, one would potentially be able to extend the features of the client. Second Life also offers the possibility of having collaborative sessions as multiple users can engage in tasks at the same time.

However, one would preferably use private islands for implementing the tools and scenarios we have in mind, and that would become prohibitively expensive. We would prefer that the requirements of the virtual environment are as low as possible, and a Second Life-based approach would require users to have broadband internet access which is still an expensive prospect in South Africa and many developing nations.

We concluded that on-line virtual environments were not suitable for our purposes. The cost of internet access along with acquiring private regions in the virtual worlds can be prohibitively expensive.

In this chapter we discussed the existing freely available open-source game engines that

could be used as a major component in the virtual environment platform we envisage. Furthermore, we discussed some of the past and existing research on using virtual environments for ASD therapy. We found that virtual environments offer potential as tools for ASD therapy and that research in this regard is still active. In the following chapter we discuss the goals and the design of our 3D virtual environment platform.

Chapter 3

The Design of the Three-Dimensional Virtual Environment Platform

3.1 Introduction

In this chapter we discuss the overall design of the 3D virtual environment platform that we developed. We begin the discussion by elaborating on the goals of the project, turning these high-level goals into a number of lower-level systems that ultimately constitute our platform. We then discuss and motivate the design decisions behind each segment of the larger platform.

We embarked upon the design and implementation of a generic 3D virtual environment platform with the following goals. The platform must:

- be able to manage and render 3D virtual environments,
- be adaptable and extendable for specific needs,
- have longevity: it should remain useful in the future,
- be well documented,
- have a generic input management system,
- have a customisable graphical user interface (GUI) system,
- include a replay system,
- have a data capturing system,
- provide a remote monitoring facility where therapists can monitor a session over a network,

- provide multi-lingual support in games and therapy tools through a translation management system, and
- provide a framework upon which tools can be built to aid non-programmers in constructing educational games and ASD therapy tools.

From these goals it is obvious that our project requires interactive 3D virtual environments and thus requires a library for creating and managing such environments. The most common example of real-time interactive 3D virtual environments is seen in modern 3D computer games. In the modern video game industry games are created with reusable middleware libraries, typically called game engines [17]. In our case we are concerned with interactive 3D virtual environments and the creation of educational games and ASD therapy tools. Thus we would require middleware similar to a game engine as part of our proposed platform. In order to compile a more complete list of requirements, we briefly consider the typical functionality provided by game engines. An individual playing a game must be able to provide input to the game and receive output from the game. Output consists of real-time rendered 3D graphics, virtual characters, animations, audio and text. Another aspect to the output provided by a game is that of realism: certain games aim to create authentic and immersive experiences. With the increase of computing power it has become common for games to utilise a physics simulation engine to improve the simulation of the virtual world that the game presents to the user. In our case the design and goals of each individual game or tool would determine the applicability of a physics simulation, but we would almost always require basic functionality such as collision detection.

A player interacts with a game through input peripheral devices such as keyboards and mice. In the context of our project we would require a fairly robust and generic input system that would make it possible to add support for many kinds of input devices: certain individuals with ASD also struggle with fine motor skills [26, 27] and would benefit from more advanced input mechanisms, such as speech recognition.

It has become common for game engines to provide tools, for use by game developers, to create levels or maps (although there is some debate over whether such tools are to be considered part of the game engine or not [7]). In recent times this would include scripting functionality: portions of the engine library would be exposed to a high level language. Scripting languages have some advantages over compiled languages since it can eliminate the time it takes to recompile game-specific code, as well as providing a very definitive manner for separating the game engine from a specific game implementation [17, 42]. When a level in a game is created, scripting can be used to implement portions of game logic, for example.

It is therefore clear that game engines consist of, and have to manage, a number of subsystems. Furthermore each of these subsystems could have a temporal dependency.

An example of where such a temporal dependency exists would be during audio playback. Consider a task that is repeatedly executed and that is responsible for playing an audio track. Such a task needs to know the time between executions so that it can sample the correct amount of audio data during each update in order for the audio track to be played at the correct speed. Another example would be the physics simulation: a box that is falling, due to gravity in the virtual world, must fall with the correct velocity and acceleration. The physics system would need to be periodically updated with an accurate time step value to correctly simulate the falling box. Along with the potential temporal dependency, it must appear to the player that the various subsystems in the engine are updated simultaneously and continuously: graphics rendering, animations, audio and input from the player must appear to all be occurring continuously and at the same time. This leads us to consider that game engines could have a process or task based structure, similar to the process management structures found in operating systems: a number of processes appear to be executing simultaneously but are actually executed in turn with certain time intervals [24]. Of course, with the advent of multi-core CPUs, it is indeed possible for more than one process to execute simultaneously, but if anything it leads to more complex process management. It seems that such a task based structure is not uncommon: the Enginuity Engine [22], discussed in Section 2.2.3 and the Panda3D engine [61], discussed in Section 2.2.4, use such a structure.

Game engines generally also include functionality to manage entities. Entities, sometimes called game objects, are objects in the virtual world (such as tables or chairs in a room in the virtual world) rendered and managed by a game engine. In object-oriented programming language terms these entities are represented by classes from a class hierarchy. Generally these classes would be divided into broad categories, such as moveable and non-moveable objects, where each class instance would be assigned a 3D model for its visual representation (if it indeed has a visual representation). In other words, there would not necessarily be different classes for a table or chair in the virtual world: both could be considered moveable objects. The class hierarchy is then used by game developers when creating objects for the levels or maps in a game. The entity management system in the game engine would be responsible for the creation of instances of game object classes, memory management throughout the lifetime of an instance, and destroying instances. Such a system is common in game engines and can be seen in OGE (discussed in Section 2.2.2), where it is called the object system, and in Crystal Space (see Section 2.2.5), where it is known as the Crystal Entity Layer.

We discussed the requirements of our platform, as it relates to educational games and ASD therapy tools, with therapists and educators and a number of points came to light [11, 23]. Therapists indicated that it would be invaluable if the platform could capture data about the actions that users take in the games and virtual worlds that would be created with

the platform. Upon further discussion the nature of the data capturing was divided into two categories: general event capturing and replays. With general event capturing, any game or tool would capture the relevant actions taken by a user, such as answers given to questions posed in a game. The replay system would be similar to replays in sports games or real-time strategy games. These replays would record an entire session of a user engaging in a game or therapy tool. Therapists would then be able to view these replays at a later stage to allow them to see the precise manner in which a child or patient interacted with a game or therapy tool. This implies that our platform must have support for data capturing, where the nature of the data to be captured is defined by each individual game or tool, and support for replay recording.

Therapists also indicated that remote monitoring and control would be a desirable feature. This entails a therapist connecting from her computer, over a network, to the computer of a child or patient engaged in a therapy tool. The therapist can then monitor the progress of the child in real-time, as well as potentially pause the game if she wants to explain something to the child.

Another important factor, and something that is not uncommon to game engines in general, is that of support for highly customisable graphical user interfaces. Some games or therapy tools implemented with our platform will require that users give certain input through some form of GUI: consider a simple educational game that poses a number of questions to a player and requests answers from the player. Although one would like to disguise this process in game play elements as much as possible, it is conceivable that some games or tools would rely heavily on a GUI component. Furthermore, individuals with ASD can be easily distracted by poorly designed GUIs [36], hence the customisability of GUI components are important.

We have extended the basic requirement of middleware similar to a game engine into the following, more specific, requirements of our platform:

- visual output, such as rendering, animations and scene management,
- collision detection or possibly support for a more complete physics simulation,
- audio output,
- generic input handling,
- development tools (such as map editors),
- a scripting language interface,
- a task management interface,

- entity or game object management,
- a data capturing system,
- a replay system,
- networking support, and
- a customisable GUI system.

We now briefly recap our conclusions on existing freely available open-source game engines and their respective advantages and disadvantages, as discussed in Section 2.2.1–Section 2.2.7. We started our investigation by considering the Quake 3 engine. This engine is not appropriate for our needs, due to the lack of documentation and instructions for its use, along with the age of the engine. In Section 2.2.4 we considered the Panda3D engine. The engine has many of the features that we require, but through our own experimentation we have some performance concerns. Deploying and using the engine on older hardware would be problematic. We considered another open-source engine, OGE, in Section 2.2.2. At the time of writing the engine was not functioning and is undergoing a complete redesign, therefore we cannot use OGE. In Section 2.2.6 we considered the Blender game engine. A lack of complete documentation along with no existing in-game GUI support are restrictive factors. We discussed the Enginuity Engine in Section 2.2.3. Its design gives an excellent insight into the management structures of a game engine, especially that of task management. However, the engine lacks a strong graphics engine implementation and lacks existing support for GUIs. We also have some concerns about the stability of the engine since it is not an active open-source project that is continually being developed. The Crystal Space engine, discussed in Section 2.2.5, is a feature rich engine. It is also one of the few engines in our survey that supports a plug-in framework for extending the functionality of the engine. At the time of writing, Crystal Space has the following notable issues: only OpenGL is currently supported for hardware based rendering and it does not appear to be widely used in large scale projects. Even though the engine has existed for a number of years, its featured projects page is not impressive. Also, there does not appear to be frequent new releases or a large scale plan for future development and feature extensions. The final engine we considered is OGRE, in Section 2.2.7. OGRE is an engine that is simple by design: it has a compact and well documented API along with support for feature additions and extensions through a plug-in API. Although OGRE is a graphics rendering engine and not a game engine, the OGRE community has created numerous plug-ins and add-ons that can add most of the required game engine features to the engine. Therefore we use OGRE as a major portion of our 3D virtual environment platform. The advantages of OGRE are discussed at length in Section 2.2.7, but our primary motivations for using OGRE are:

- It is a graphics rendering engine with support for scene management through a scene graph structure. There exists a number of plug-ins that add support for different kinds of scene graph implementations that can be utilised for better results in certain situations. For instance, there is an octree based scene graph implementation for general use, a height map based implementation for terrain rendering and a paged landscape based scene manager allowing seamless travel on a large landscape.
- It supports scene animations and skeletal animations for realistically animated virtual characters.
- The API is designed to be simple and compact and is well documented. This relates directly to the similar requirement we have of our platform.
- Our empirical tests showed that OGRE is among the best performing engines that we surveyed.
- It supports extensions through a plug-in interface.
- OGRE can manage positional sounds through a plug-in.
- There are four existing plug-ins that add physics and collision detection support to the engine.
- There are two mature add-ons for GUI management and rendering in OGRE, both with WYSIWYG editors for GUI layouts.
- There are existing libraries for OGRE in some higher level languages, most notably pyOGRE for Python.
- OGRE has been used in a number of commercial projects such as *Ankh* and *Jack Keane* by *Deck13 Interactive* [5] and the upcoming *Torchlight* by *Runic Games* [6]. OGRE has an impressive list of projects on its featured projects page [54].
- Although Crystal Space does provide some of these features as well, the simpler design, better documented API, more active community and large variety of plug-ins and add-ons makes OGRE the better option, in our opinion.

We would have to implement a number of features that are not provided by OGRE or its associated plug-ins and add-ons. These features are:

- task management as seen in EE and Panda3D;
- game object management, that is, a hierarchy of classes that is used to represent objects in the virtual world. OGRE has basic graphical entity management, but we shall have to extend this so that entities can have more associated information and be more extendable;

- replays;
- data capturing; and
- networking.

None of the engines we surveyed provide data capturing support. Some engines, notably Panda3D and Crystal Space, do provide some networking support (generally socket management and basic messages) but no complete networked game session management. Panda3D does provide a recording system that could be used for the basis of a replay system, but none of the other engines provide such functionality. The Crystal Space engine does provide the so-called Crystal Entity Layer that adds support for entity or game object management to the engine. However, using OGRE and implementing our own task management and game object management systems is preferable since:

- it provides one with more control. If we use the Crystal Space implementation, we have to use their implementations where if we use OGRE we could implement more relevant and compact solutions for our situation, and
- it allows us to design a smaller, well documented and easy to use API that relates to our specific requirements.

In this section we discussed the high-level goals of this project and extended those goals into a set of lower-level requirements of our 3D virtual environment platform. We use OGRE as a major component in our platform, since it provides us with the best balance of features, extendability, design and stability among the well known open-source engines. In the remainder of this chapter we discuss the design of our 3D virtual environment platform in depth. Each following section is dedicated to a particular component or feature of the platform. We begin the discussion by focussing on the design of the task management interface.

3.2 The Task Management Interface

Virtual environment engines consist of a number of subsystems, where each of these subsystems could have a temporal dependency. However, it must still appear to the player that the various subsystems in the engine are updated simultaneously and continuously. This leads us to consider that game engines could have a task based structure, similar to the process management structures found in operating systems: a number of processes appear to be executing simultaneously but are actually executed in turn with certain

time intervals [24]. Both the Enginuity Engine [22], discussed in Section 2.2.3, and the Panda3D engine [61], discussed in Section 2.2.4, have task management structures.

Our task management interface must include a hierarchy of classes representing different types of tasks. One reason for this is that each subsystem (such as rendering, input and audio) in the engine requires a specific task implementation. Also, one of our goals is to make our engine as extendable as possible. We aim to achieve this by including a plug-in API. The plug-in API is discussed in more detail in Section 3.4, but essentially plug-ins would often add new tasks to the engine. Thus in order for our task management system to support such a plug-in system, and for our design to be versatile, our tasks must be extendable through inheritance. Furthermore, there must be a single, well-defined mechanism through which tasks are managed, updated, added and removed from the engine. Therefore, we require some class to act as the base class of all tasks and a class that acts as the manager of all tasks in the engine. This design, motivated especially by the extendability requirements of our platform, is similar to the one used in the Enguinity Engine (EE) [22] discussed in Section 2.2.3. We now consider whether OGRE can provide us with the basis for such a task management system. OGRE has a frame notification system, where functions in a class can be called at the beginning and/or end of a rendered frame. In order for a class to receive frame notifications, it must extend an appropriate OGRE base class, namely the OgreFrameListener class. Therefore, it would be possible to define specific tasks through inheritance from the OgreFrameListener class. OGRE also includes the time between frames in calls to frame listeners. However, the frame notification system of OGRE was not designed with the full functionality of a task management system in mind. OGRE does not allow one to define custom time intervals in which to update frame listeners; it always updates all frame listeners before and after every frame. In Section 3.2.1 we discuss why we would not necessarily want to update all tasks as often as possible. Therefore, we cannot simply use the frame notification system of OGRE for our task management interface.

We can consider a management system such as the one used by the OpenGL Utility Toolkit [35] (GLUT). In GLUT a function is assigned for every subsystem. For instance a function is assigned for rendering, and a different function is assigned to handle input. Such a system is not suitable in our case, since it is not as extendable as we require, due to the fact that in GLUT the subsystems are generally predefined.

From the preceding discussion it is clear that we must design our own task management system. As in the EE, our system will consist of a kernel that manages a set of tasks, but in addition we include support for multi-threading. This implies that certain tasks will execute in their own threads, separate from the kernel thread. We feel it is important to include multi-threading in the design, since this engine is part of a long term project and currently, in the field of game engines, there is a shift from single-threaded engines

to multi-threaded engines [62]. By including support for multiple threads in the engine, we position the engine for potential use in future research on multi-threaded game engine architecture, as well as simply enabling the engine to potentially better utilise multi-core CPUs that are becoming more prominent. Furthermore, there is an advantage to certain engine systems executing in their own threads. If the engine encounters a performance issue during, for example, the update process of the physics task, then other tasks that are running concurrently with the physics task will not necessarily starve.

We call our envisioned task management class the Kernel. Each of the subsystems of the engine that require repeated updates or has a temporal dependency, should be represented by a task. For example, the engine should contain a video task that renders a scene, a physics task that manages a physics simulation for the virtual world, an audio task that manages audio playback and an input task that is responsible for processing input received from the user. The task hierarchy is shown in Figure 3.1. This hierarchy should have a base class, called the Task class. This allows us to define the functions in the Kernel interface to receive references to Task instances, but through inheritance and polymorphism it would then be possible to use any class in the kernel which was derived from the Task class. The basic Task class should be defined to be updated in the kernel thread, that is, it does not have its own thread. We can then extend the Task class with a new class, namely, the ConcurrentTask class. The ConcurrentTask would use the Runnable interface from the Portable Components[1] (POCO) [41] C++ library to enable it to run in its own thread. Plug-ins that require their own unique classes can then define a class that extends from the Task or ConcurrentTask classes, depending on whether the task should be updated in its own thread or not.

In the conceptual role of the kernel there should be only one Kernel instance that exists in the engine. It would be unnecessary to have more than one kernel, where each has its own set of sequential tasks: if one wants to use multiple threads to execute tasks, one could simply use concurrent tasks or extend the kernel to assign threads to certain sequential tasks as necessary. Hence, the kernel is an example of a singleton class. A singleton is a class that has no more than one instance in existence and this instance must have a well-defined global access point. See Section A.1 for more details.

Now that we have motivated the broad design of the task management interface, we discuss the functionality of the kernel in more detail.

[1] POCO is a C++ utility library. It provides one with cross-platform thread management, networking, XML parsing and many other features.

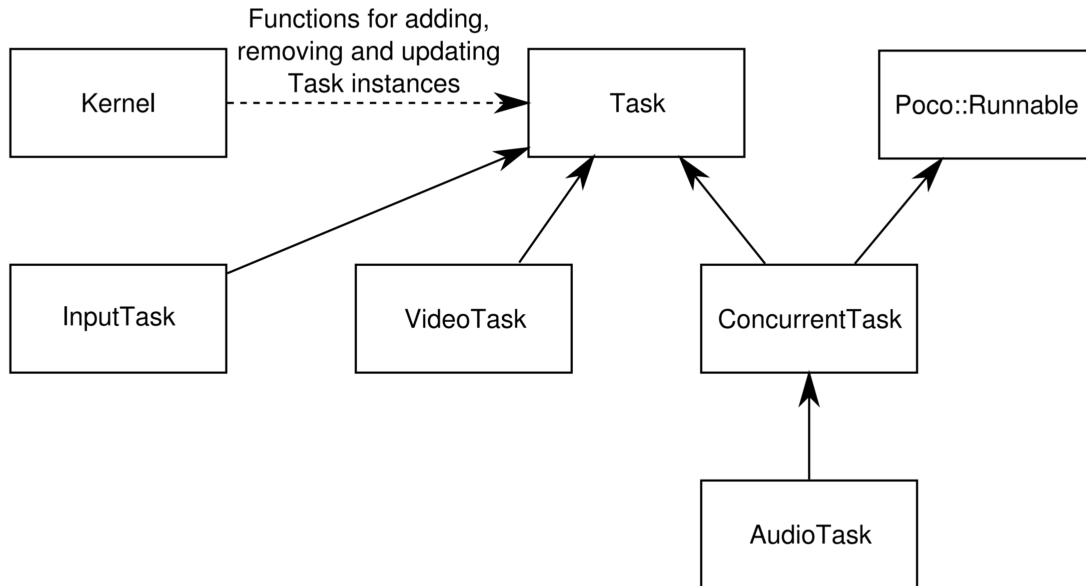


Figure 3.1: The Task class hierarchy.

3.2.1 The Functionality of the Kernel

We begin our discussion around the functionality of the kernel by discussing the similarities between our kernel and the process schedulers of operating systems. We then discuss the algorithm that our kernel should follow when executing its list of sequential tasks. We also discuss some complicating factors, introduced by the multi-threaded design of the engine, that influence the kernel algorithm. Finally we discuss the importance of maintaining a record of the time that passes between task executions as well as the execution frequencies of certain tasks.

The task management interface of our engine definitely has clear similarities with process scheduling in operating systems. Both manage task execution so that tasks appear to execute simultaneously. Operating systems accomplish this through various process scheduling algorithms and process pre-emption[24]. An operating system executes a process for a small amount of time, then pre-empts that process and executes another for a small amount of time. The manner in which the operating system decides which process to execute when a process is pre-empted, depends on the scheduling algorithm that is employed. When a process is pre-empted, its state is saved and at some point in the future the scheduler will schedule the process to continue its execution from where it was pre-empted previously. One of the more basic scheduling algorithms that operating systems employ is the round-robin scheduling algorithm [24]. This algorithm schedules each process in the system for execution, one after the other, where the order is determined by a priority associated with the process. Each process is executed for the same amount of time before it is pre-empted and another process scheduled. The similarity between operating system processes schedulers and our kernel leads us to consider whether we can

utilise scheduling algorithms from operating systems in our kernel. There is, however, one significant issue which makes the use of pre-emption difficult in game engines. The tasks found in game engines have significant linear dependencies on one another. Consider, for example, the task responsible for rendering the virtual world. It can only render the scene once it knows that all objects in the virtual world are in their final positions for the particular frame that it must render. If the physics task is busy updating positions of objects due to the physics simulation and is then pre-empted so that the video task can execute, it would be problematic. The video task would not be able to render all objects yet, since they are not in their final positions for the frame. The same issue can be found between a task responsible for input from the user and the physics task. If all of the user input for a frame has not yet been processed, the physics task cannot start its calculations for that frame. Therefore, we do not use pre-emption in our kernel, due to the serial dependency among tasks. We use simple round-robin execution where each task has an associated priority that is used to determine the order of execution. We also include two more values for every task, namely, the execution interval and the iteration limit. These values are used to determine the minimum time between executions of a task and is useful for tasks that we do not want to execute as frequently as possible[2] [17]. We discuss the execution interval of a task in more detail in the latter part of this section.

We base the design of our task management interface on that of the EE. However, we add two features to the design of our kernel that is not found in the kernel of the EE:

- support for multi-threaded tasks, and
- an execution interval for a task that determines a minimum time between executions of a specific task.

Our kernel should manage and update all sequential tasks that have been assigned to it. When a new task is created, it is assigned to the kernel which then has to initialise it and execute it when appropriate.

There are a number of tasks that will be common to many applications that utilise the engine. Among these are:

- the timer task that records the amount of real-time between frames and is used to control many actions in the virtual world (for example, a character moves at a certain speed per second, and not a fixed speed per frame, to ensure that it covers the same distance in one second on slower and faster hardware),

[2]Some tasks gain nothing from running as frequently as possible. For instance, there is no real need to run game logic more than a set number of times per second. This is discussed in more detail in the latter part of the section.

- the input task that receives input messages generated by hardware such as keyboards, mice and game pads, and
- the video task that is responsible for rendering the graphics that represent the virtual world.

The kernel repeatedly executes a number of actions, including executing its sequential tasks, until it receives a message (communication in the engine is discussed in Section 3.2.2) that it is to shut down. Algorithm 3.1 shows the algorithm that the kernel executes. The execution of receiveQueuedInternalMessages() delivers all of the kernel messages, that were sent since the last update, to the kernel. These messages are processed by the kernel and could include messages to add tasks to the kernel, remove tasks or to shutdown the kernel.

Since the engine is multi-threaded, one has to take into account that changes could be made to the list of tasks that the kernel manages while the kernel is executing a task. This could cause situations where the kernel executes a task that has been removed or suspended. There are two ways to solve this problem: using mutual exclusion locks or by duplicating the list. By using mutual exclusion locks, one would prevent any changes to be made to the list of tasks that the kernel manages while the kernel is executing a task. This does, however, reduce some of the concurrency in the engine: the thread that tries to make this modification to the list would now have to wait for the kernel thread to finish with its action on the list. We can improve on this situation by distinguishing between different types of mutual exclusion locks, one type for reading and one for writing. Thus we could allow any number of readers concurrent access to the list or we allow only one writer with no readers to access the list. This would eliminate the time that a thread that wants to read information from the task list has to spend waiting. However, if a thread wants to modify the task list it would still have to wait until the kernel releases any lock it has on the list.

The second option (duplication of the list) is useful only if the task list is relatively small or if we know that one of the threads needs access to the data structure for an extended period of time. We could then duplicate the list of tasks, and allow one of the threads to use the duplicate. In the case of the kernel we would let the kernel thread execute tasks from the duplicate and let other threads update the original list of tasks. Thus the createTheListOfTasksToExecute() method from Algorithm 3.1 returns a duplicate of the list of tasks the kernel needs to execute. The kernel uses this duplicate list while it is executing the tasks.

From Algorithm 3.1 we note that, upon each execution of a task, the kernel passes the task a parameter that contains the amount of time that has passed since the previous set of executions. This time value is determined by the timer task. It is always the first task

```
running ← true
while running = true do
   receiveQueuedInternalMessages();
   list ← createTheListOfTasksToExecute();
   timeDelta ← timeSinceThePreviousExecutionLoop();
   timeAtStartOfLoop ← absRunningTime();
   for all t in the list of tasks do
      timeSincePrev ← timeAtStartOfLoop - getTimeLastExecuted(t);
      i ← 1;
      while (timeSincePrev ≥ getExecutionInterval(t)) & (i ≤ getIterationLimit(t)) do
         if i ≠ getIterationLimit(t) then
            execute(t, getExecutionInterval(t));
            setTimeLastExecuted(t, getTimeLastExecuted(t) + getExecutionInterval(t));
         else
            execute(t, timeSincePrev);
            setTimeLastExecuted(t, timeAtStartOfLoop);
         end if
         timeSincePrev ← timeSincePrev - getExecutionInterval(t);
         i ← i + 1;
      end while
      if i > 1 then
         setTimeLastExecuted(t, timeAtStartOfLoop);
      end if
   end for
   running ← stillRunning();
end while
```

Algorithm 3.1: The design of the main loop of the kernel.

to be executed, and it is the task with the highest priority in the engine. It maintains a timer that determines the time since it was last updated, along with the amount of time that the engine has been running. Time is of the utmost importance in an engine. Since computers with hardware of differing levels of performance will not be able to perform all of the computations and graphics rendering at the same speed, all computers will not have the same number of task executions per second. Thus it is important that each task always knows the amount of time that has passed since its previous execution so that it can scale all appropriate actions with this time value. This ensures that the virtual world simulation runs at the same speed, in real-time terms, on all computers that meet some minimum hardware requirements.

An important aspect of Algorithm 3.1 is that the same time value, calculated before the loop of tasks updates, is passed to each task. This means that each task does not actually get the exact time since its previous update, instead the tasks all get the time between each successive set of task executions. Effectively each task is passed time as if it is the first task to execute in every set of executions. This is important in game engines, since we are outputting the game state (a combination of a rendered frame of graphics and audio

output, for example) in discrete steps. The amount of time that tasks have processed must be synchronised so that no task has processed more time than any other task when we output the game state in each step.

Algorithm 3.1 also shows that the kernel considers the amount of time that has passed since the last execution of a task, and compares it with a so-called execution interval of the task before it executes the task. The execution interval determines the amount of time that has to pass before a task is executed again. Consider a task that has expensive computations, such as a task that does physics computations. Since such a task would require a fair amount of computation time, one would determine a reasonable frequency for its execution. It cannot execute too often, as that could cause starvation where this one expensive task uses all of the computation time. On the other hand, it also must not execute too infrequently, as that could cause quality problems in the virtual world simulation.

The iteration limit of a task is used to determine if a task should be executed repeatedly in one iteration of the kernel loop. This is only used if the amount of time since the last execution of a task is a multiple of its execution interval. This type of iteration is necessary if a rarely executed operation, such as paging to the hard drive, caused a slowdown in the engine. Repeated execution of a task allows it to remain synchronised with the real amount of time that has passed. One could argue that a task could simply be executed with this large change in time value, instead of repeatedly executing it with its smaller execution interval as time value. This may well work for certain types of tasks, but it is largely dependent on the type of actions or computations that a task performs. If we return to the example of a physics task: the mathematical models that most physics engines use, rely on being solved in repeated iterations with small time steps. The iteration limit ensures that the execution of computationally expensive tasks do not cause even more performance problems. In [17] a similar timing model for use in game engines is discussed. Many of the ideas presented in that timing model has been integrated into our design.

We have discussed the kernel and its set of tasks. We have also discussed the use of concurrent tasks and multiple threads. We have not discussed, however, if and how tasks communicate with one another. In the next section we motivate why a communication system among tasks is required. We then discuss possible designs and motivate the system we ultimately implemented.

3.2.2 Communication

Communication in the EE takes place simply through direct function calls among tasks. This form of communication is not desirable, since there is no well defined structure

that governs the communication. When new tasks are added, or existing tasks extended, implementers will not know how to safely communicate events and information among tasks. In order to address this issue, and improve the general extendability of our engine, a well defined engine-wide communication system is required. Furthermore, the introduction of concurrent tasks in our engine further emphasises the need for a well defined communication system. Calling functions directly among concurrent tasks can complicate thread-safety dramatically. It is much simpler, safer and more convenient if some form of message could be sent from one task to another, at an appropriate time, instead of tasks repeatedly calling functions among one another to check progress or fetch information. We started the discussion of the communication system by focussing on communication among tasks. It would, however, be advantageous if this communication could be extended for general communication throughout the engine, so that senders and receivers of messages are not necessarily limited to tasks. Consider, as an example, the input handling system of the engine. It could use the communication system to notify any interested parties about input events. Plug-ins that extend input functionality, or applications built using the engine are examples of parties that would be interested to receive communications about input events. Thus our primary motivation for including a communication system in the engine is:

- tasks could be dependent on one another and require periodic information from one another,
- it is simpler, safer and more convenient than calling functions among concurrent tasks, and
- with a well defined communication API in place we increase the extendability of the engine, since plug-in implementers will know exactly how to safely facilitate communication.

As the tasks and kernel of the engine are analogous to the processes and kernel in an operating system, so is the communication system of the engine analogous to inter-process communication (IPC) in an operating system [24]. Two of the most popular techniques for IPC is that of shared memory among processes and message passing. We briefly discuss both approaches and then motivate the system we use in the engine. In operating systems, when shared memory is used for IPC, it is common for a shared memory block to be mapped into the virtual memory space of each of the two communicating processes. When communication between the processes must take place, one of the processes will write data into the shared memory block and the other can then read data from it. It is obvious that in a multi-threaded system care must be taken to ensure that two processes do not access the same segment of the shared memory block in a dangerous manner. It

would be safe for multiple processes to read from shared memory simultaneously, but if a process is writing no other processes should access the shared memory block. In the case of a message passing system, there is generally some message passing API available in the kernel. Processes can fill a message with data and send it to another process. This requires message queues to be set up, so that incoming messages for a process can be queued until the process has an opportunity to receive these messages. An advantage of shared memory over message passing is, the fact that all communicators share the data that is being communicated. In message passing a message object is created and sent to receivers. This requires the message object to move through some form of messaging system and could involve queuing and duplication (if it is delivered to multiple receivers). Thus message passing has certain size overheads involved in message transmission that it is not present in the shared memory model. Notwithstanding many modern operating systems, for example the Mach microkernel in Mac OS X, use message passing [24]. The primary advantage of message passing over shared memory is that it is simpler and safer to use. Furthermore, since we are aiming for an extendable engine, we must ensure that our communication model is built on an extendable base. Message passing provides us with the potential for building a hierarchy of message types that can be extended: we define a generic Message base class in our engine and any specific type of message inherits from it. A further advantage of using message passing, in our case, is that it is simpler to use for communication over a network[3] [24]. Thus our primary motivation for using a message passing system is:

- the concept of messages that are passed among objects allow us to build a hierarchy of message types that extend one another. This improves the extendability of the engine, as it allows for plug-ins to follow a well defined structure for creating new message types;
- it is simpler to use message passing to communicate over a network; and
- one does not have to set up shared memory among communicators and manage the thread-safety of the memory block.

Message passing does have a size overhead involved in the sending and queuing of messages, something that is not present in shared memory, but it can be addressed to some extent. We discuss how we can address these overheads in message passing in Section 4.2. We now discuss the design of the communication model that our engine uses.

The message-passing model we employ consists of a notification manager, various message queues and various message types. We designed our system on top of the message-passing

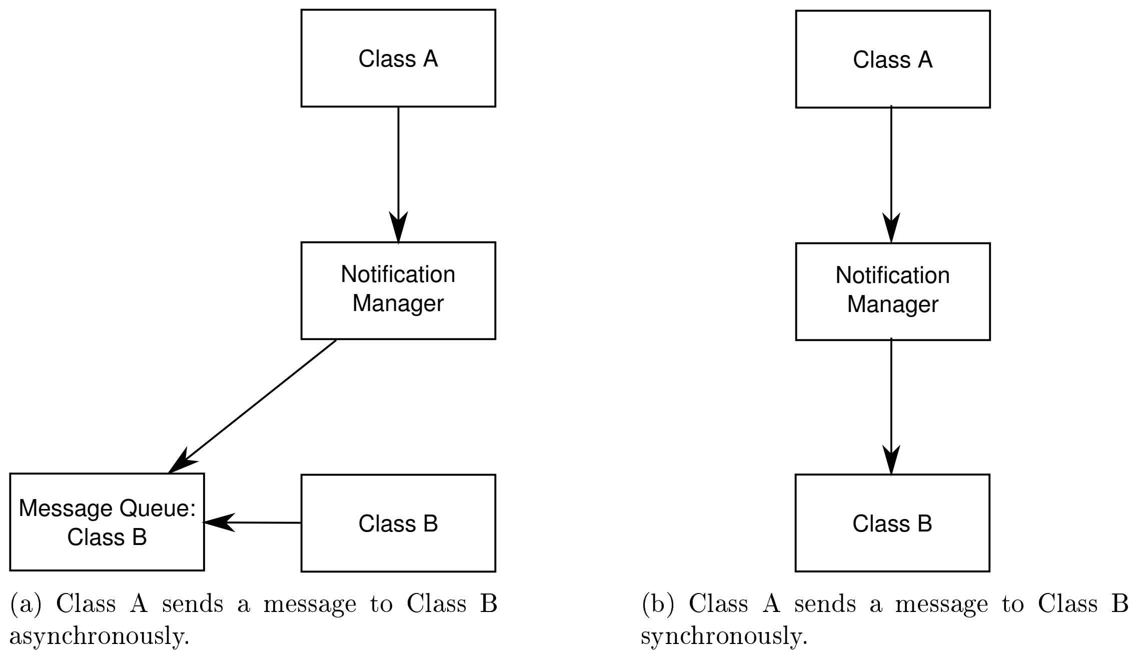
[3] We discuss using message passing over a network in the design of our networking system in Section 3.15.

system provided by the POCO library, since it provided some of the functionality that we required and is thread-safe. A notification manager is used to determine to which message queues to deliver a message of the specified type. Generally every object or instance informs the notification manager of all the message types it wants to receive and it is then assigned a message queue by the notification manager. When an object registers itself as a recipient of messages, it must provide methods for each of the different types of messages that it wants to receive. The notification manager delivers all sent messages, of a certain type, to all objects that have registered to receive messages of that type.

It is not uncommon for the message passing system employed in operating systems, for inter-process communication, to support both synchronous and asynchronous delivery [24]. In our engine we have sequential tasks, that are executed in a specific order. It makes sense, in this case, to use an asynchronous message passing model: messages sent to a task are queued and processed when the kernel executes the task in question. It also makes sense to have a synchronous delivery model: the concurrent tasks in our engine might be working in a cooperative manner and could require synchronous delivery of messages. For example, one of the tasks may have to wait for an action from another task to complete before continuing. Synchronous delivery of an update message would be more appropriate in such a case.

The notification manager supports delivery of messages in both a synchronous and asynchronous fashion. The asynchronous method uses the message queue assigned to an object. The synchronous method does not use the message queue and delivers the message to the receiving object directly by using the notification manager to find the appropriate registered function for the receiving object. In the case of asynchronous message delivery, it is the responsibility of the receiving object to periodically request the notification manager to deliver the messages waiting in its queue. Figure 3.2(a) illustrates the asynchronous method and Figure 3.2(b) illustrates the synchronous method of the message-passing model that we employ.

One potential issue with this design is performance issues related to the creation and destruction of numerous message instances during run-time. This can be especially problematic if large numbers of messages are sent at the same time. We address this problem in two ways. The first is that, if the same message is to be sent to multiple receivers, we do not create cloned instances but instead share the message among its receivers and only destroy it once it is no longer required. The reference counting mechanism provided by the POCO library is used to determine when the message should be destroyed. The second manner in which we address the potential performance concerns is through the use of so-called object pools. These pools pre-create a number of message instances. When a message is to be sent, a new instance is not created but instead fetched from the pool. We discuss the reference counting mechanism and the object pooling system in more detail



(a) Class A sends a message to Class B asynchronously.

(b) Class A sends a message to Class B synchronously.

Figure 3.2: Class A sends a message to class B via (a) asynchronous and (b) synchronous message passing.

in Section 4.2.

In this section we discussed the message-passing based internal communication system employed by the engine. The message-passing system is also (as with the kernel and its associated kernel manager) managed by a so-called manager class called the notification manager. In the following section we briefly discuss the concept of manager classes as it is applied in the engine.

3.3 Managers

In this section our discussion focuses on the use of so-called manager classes in the engine. During the design of the engine we identified the following classes which appear quite frequently:

1. an extension of the basic task class,
2. a message type associated with the specific actions of the task, and
3. a class that provides an API for users of the engine to control certain actions of the task.

From this pattern we concluded that it would be advantageous if we were to add some structure or convention to the class that provides an API for control over the task. We

called these classes manager classes. Manager classes generally have numerous methods to allow control over a task. The majority of these methods would send messages to a task, informing it of the actions it should take.

It also became apparent that users of the engine would need a global and well-known access point to each and every manager instance. Moreover, only one manager instance is necessary per task type: for the majority of task types, there will be only one active instance. For those tasks that could have numerous multi-threaded instances, one would still require only one manager to control the set of instances. Thus the manager class concept satisfies the conditions of being a singleton, as stated in definition A.1.1, on page 112. All manager classes in the engine are therefore singletons.

In the following section we discuss the mechanism whereby engine functionality can be extended through so-called plug-ins.

3.4 The Plug-in API

One of the goals of this project is to implement an engine that is generic and extendable. While the engine is open-sourced and thus allows anyone to familiarise themselves with its internal functionality and extend it, we aim to add a well defined manner in which functionality can be added through the implementation of plug-ins. The use of plug-ins to extend the functionality of a program is well-known and can be seen in environments such as Eclipse [2] and Netbeans [3]. Also, its use is not uncommon in engines, for example, both OGRE (discussed in Section 2.2.7) and Crystal Space (discussed in Section 2.2.5) support plug-ins. Therefore, our primary motivation for including a plug-in API is to allow for the engine to be extended with specific functionality without requiring modification of the internals of the engine. An example of where this could be useful is a plug-in that adds extended input device support, for instance support for speech recognition.

In order for the plug-in system to be effective, there must be well defined ways in which it can extend engine functionality. The manner in which a plug-in API for a specific piece of software is implemented largely depends on the design of the software. In our case concepts from object-orientation such as inheritance and polymorphism are central to the design of our engine and its extendability. Therefore our plug-in API extends engine functionality by providing implementations for existing abstract class hierarchies. An example is the audio system of the engine, as discussed in Section 3.10. The audio framework consists of a number of abstract classes that must be implemented by plug-ins to enable audio output through a specific audio library. This plug-in design is similar to the design found in OGRE. It is also possible for plug-ins to add completely new functionality without extending existing abstract class hierarchies. When a plug-in is

implemented, it is possible that certain initialisation steps are required. A plug-in could define a new task and would need to instantiate it and add it to the kernel. Later it would need to potentially remove the task from the kernel and destroy it. Plug-ins therefore would benefit from well defined initialisation and destruction steps.

We base the design of our plug-in system on that of OGRE. We define an abstract Plugin class, based on the one provided by OGRE, that has a number of virtual functions:

install(): this function is called when the plug-in is installed in the engine, but before the initialisation of any subsystems in the engine,

initialise(): called just after the engine has been initialised,

shutdown(): called just before subsystems in the engine are shut down, and

uninstall(): called after all subsystems in the engine have been shut down.

Plug-ins must extend the Plugin class and implement these four functions. Plug-ins would generally be included as shared libraries in the resulting application. In that case the plug-in library must also provide a function named DLLPluginStart(). In the Application class of the engine, which acts as the entry point for an application that uses the engine, there are functions for loading and unloading plug-ins. These functions take the name of the plug-in as a parameter and then try to dynamically load the shared library. Once the library is loaded, the function searches for the DLLPluginStart() function in the library and executes it. Generally the DLLPluginStart() function creates the Plugin instance of the plug-in and from there the Application instance calls the functions in the Plugin instance as appropriate. In Chapter 4 we discuss some of the plug-ins we have created for the engine in more detail. Among these are a plug-in for sound playback and a plug-in for loading levels or scenes that have been exported from Blender.

In the following section we discuss inclusion of collision detection and physics in our engine.

3.5 Collision Detection and Physics

With the increase of computing power it has become common practice for games to utilise a physics engine to improve the simulation of the virtual world that the game presents to the user. Examples of modern engines that include a physics engine is the Unreal 3 engine [20] and the Source engine [64]. In our case the design and goals of each individual game or tool would determine the applicability of a physics engine, but we would almost always require basic functionality such as collision detection. As an example, if a player

controls an avatar, the engine would certainly need to be able to detect if the player tries to move her avatar through a wall and stop this from occurring. In this section we discuss the inclusion of a physics engine in our platform.

As discussed in Section 2.2.7, a number of add-ons exist to add a physics engine to OGRE and therefore to our platform. These are:

OgreODE: a physics add-on that uses the Open Dynamics Engine (ODE),

OgreNewt: a physics add-on that uses the Newton physics engine,

OgreBullet: a physics add-on that uses the Bullet physics engine, and

NxOgre: an add-on that uses Nvidia PhysX to add physics support to OGRE.

All of these physics add-ons perform collision detection as well. There is also an add-on available for OGRE called OgreOPCODE, for only collision detection. It does not make sense to use an add-on that only adds collision detection if, for a similar amount of design and implementation effort, we can have a full physics engine in our platform. Therefore we now consider each of the physics add-ons for OGRE and then determine the best add-on for our situation.

ODE and Bullet are open-source physics engines. PhysX is a proprietary engine by Nvidia, but it is free to use for commercial and non-commercial purposes. Newton is not open-sourced but also free to use. These engines all have similar basic functionality for rigid body dynamics. Bullet and PhysX have support for fluid, soft body and cloth simulations as well. Furthermore, Bullet and PhysX also support multi-threaded physics solvers, allowing for better utilisation of multi-core CPUs. PhysX has support for hardware acceleration through the graphical processing units (GPUs) of a number of recent Nvidia graphics cards. We ran empirical tests that showed that Bullet and PhysX perform well even with a large number of physics-enabled objects in a scene. Furthermore, there is a visual debugging tool available for PhysX. This tool allows one to see a visualisation of a PhysX enabled scene. One can then confirm that the physics properties and collision models for all objects in the scene are correct.

We therefore use the NxOgre add-on with Nvidia PhysX as our physics solution. Our primary motivation for using Nvidia PhysX is based on the following points:

- empirical tests showed PhysX to be one of the best performing physics engines available to us,
- there are a number of tools available for visual debugging and the creation of collision models on the Nvidia website [40],

- it supports multi-threaded solving utilising multi-core CPUs and hardware acceleration through some of the newer Nvidia graphics cards, which are useful features for the long term feasibility of our platform, and
- it has support for soft-body, cloth and fluid simulations.

Since PhysX is not open-sourced, we could encounter technical support issues. Nvidia does have public on-line support forums but proper technical support is given through paid support subscriptions. If we encounter problems it might be somewhat more difficult to solve them than in the case of open-source projects with active communities. However, PhysX is commercially successful and mature so we believe this should not be a problem. Note that it would be possible to switch to another physics engine if it becomes necessary, although such a switch would require a number of changes: a physics task would have to be implemented for the new physics engine, the game object hierarchy and factories, discussed in Section 3.6, would have to be extended in order to support the new physics engine, and finally the class used for scene management, discussed in Section 3.8, would also have to be modified. Due to the nature of the scene management class, one would not be able to simply use a plug-in to replace the physics system, as it would require direct modification of the engine. This is somewhat regrettable, but these difficulties were introduced due to a trade-off among how abstract the interface is, how easy-to-use it is and how well it performs.

Physics engines, PhysX included, usually define the concept of actors and bodies. In general actors would be classes that contain most of the physics related information, such as velocities and mass. Bodies generally extend actors to include a visual representation of an object and are usually defined by the application that uses the physics engine. During our discussion of the game object hierarchy in the following section, the concepts of actors and bodies will be discussed in greater detail.

3.6 The Game Object Hierarchy

OGRE lacks a satisfactory implementation of entity management. Entities in game engines are any objects in the virtual world that are visible, renderable or can be interacted with by the user. An entity management system is common in game engines and can be seen in the Open Game Engine (discussed in Section 2.2.2, and in an extension to Crystal Space (see Section 2.2.5) known as the Crystal Entity Layer. In this section we discuss the design of an entity management system in our engine. We use the term “game object” to refer to an entity. We discuss the design of a class hierarchy of game objects, as well as the manner in which instances of these classes should be created, managed and destroyed.

The hierarchy of classes, starting with a generic and extendable base, that are used to represent the different types of objects that can exist in the virtual world managed by the engine, is called the game object hierarchy. We use such a hierarchy with an extendable base class in order to satisfy the general extendability requirement of our engine. Implementations using the engine can simply extend one of the classes in this hierarchy to create any customised object that can exist in the virtual world. Other systems and managers in the engine could be written to use a certain base class in the hierarchy – these systems would then automatically support any classes derived from this base. This again supports the generic and extendable design philosophy of the engine. There should be one abstract base class of the game object hierarchy, which we call the GameObject class. In the following section we discuss the GameObject class in more detail.

3.6.1 GameObject

The GameObject class serves as the base class for all objects that are representable by the virtual world. The class stores some general information, such as the name of the object, various customisable properties and a list of actions. We now begin with a discussion focused on the content of the GameObject class, motivating the structure of the class.

In our engine, instance names are important: these names are the property that is used to uniquely identify an instance. Thus every instance of the GameObject class has a unique name. For example, a set of GameObjects representing a number of chairs and a table in the virtual world, could be named as “chair1”, “chair2” and “table”. Our motivation for including unique instance names in the design is the fact that it provides us with an easy manner in which to find and identify individual instances. In the next section we briefly the discuss the design of the instance and class naming system used throughout the engine, our reasons for including such a system, and its merits.

3.6.1.1 Class and Instance Names

By evaluating our early design iterations, we concluded that it would be invaluable to have a structure whereby we could provide names to the instances of certain classes. Conceptually this would allow us to refer to instances by their names in certain areas of the engine, without necessarily being in possession of a reference to the instance. This facility would enable us to uniquely identify an instance, and it could be used to fetch a reference to an instance by using the instance name, as required. A mechanism that would allow us to easily know the class name of an instance we are working with, would also be invaluable. A system for defining and getting class names would be useful since our engine has many abstract base classes due to its extendable design. Thus we are often

in a situation where we have a reference to a base class, but we are calling appropriate functions in the class instance through polymorphism. A situation could occur where we would need to know exactly with what type of class we are working. A class naming system would then allow us to find the class name of the instance we are working with, at run-time, when we only have a pointer to an abstract base class instance. Similar functionality could be gained by simply using the typeid operator or successive dynamic_casts in C++. However, this would require that the engine binaries are compiled with run-time type information (RTTI) which could add size and performance overheads to the binaries. Therefore, we designed two classes that are to be extended by classes from which we require such naming functionality, namely the NamedObject and NamedInstance classes.

The NamedInstance class simply provides the mechanisms for storing an instance name. It is included once in an object hierarchy, generally by the eldest ancestor. The NamedObject class, however, must be extended by every class that requires a function that returns its class name. This can be considered somewhat problematic, since in a large class hierarchy it would be included at every level and could lead to some overhead concerns. However, the class is small, since it only has one data member, a string. Disabling RTTI and including the NamedObject class only in class hierarchies where we need such naming functionality should lead to saving over using RTTI, in our opinion.

In the next section we continue with the discussion on the structure of the GameObject class. We discuss a mechanism that allows one to store a set of properties in a GameObject instance. This is generally used to allow one to extend the information that can be stored in a class instance without using subclasses.

3.6.1.2 Customisable Properties

We are discussing the design of our entity management system, or game object hierarchy. We have introduced and motivated the use of the hierarchy, the fact that classes in the hierarchy extend one another to add new functionality and the fact that all instances of classes from the hierarchy will have unique instance names. With this model, if we want to extend the functionality or the type of information that a certain class in the hierarchy can store, we have to create a new class in the hierarchy that extends some existing class. While this adheres to our extendability requirement and uses well defined object-orientation principles, it can be overly complicated in our case. Consider the example where we have an instance of some GameObject type. Now we wish to set a property of this instance, for example, a movement speed. If there is no existing object in the hierarchy with such a property, we would now have to create a new class type. In this case, where we simply want to add a single piece of data, this seems unnecessarily complicated. Therefore we add a set of customisable properties to the GameObject class.

This allows one to set key–value properties, such as the movement speed of an object, on any GameObject class. Another reason why such a set of properties is useful, stems from the fact that we also want the engine to be usable from higher-level scripting languages. Not all scripting languages have the same level of support for object-orientation and inheritance. Customisable properties is a simpler way to extend the data that can be stored in a GameObject instance from scripting languages.

In this section we discussed the set of customisable properties maintained by GameObjects. These properties are simple strings that are mapped to values. This is a simple technique that allows users of the engine to add various types of properties to a GameObject without the need to create subclasses.

Thus far we have discussed the GameObject hierarchy and the GameObject class in particular. We now introduce two new classes in the GameObject hierarchy, namely the GameActorObject and GameBodyObject classes (see Figure 3.3 for the complete hierarchy). These classes are required for the integration of the PhysX physics library, discussed in Section 3.5, with our engine. The concepts of actors and bodies were briefly introduced in Section 3.5. In the next section we discuss actors and bodies, and ultimately the GameActorObject and GameBodyObject classes in more detail.

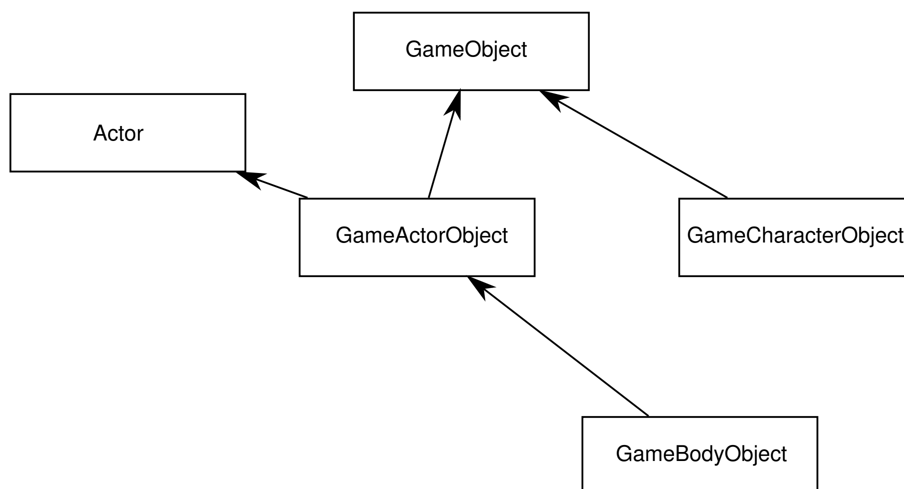


Figure 3.3: The hierarchy of classes that represent the various types of objects in the virtual world.

3.6.2 Actor and Body classes

In the hierarchy shown in Figure 3.3 we find the classes GameActorObject and GameBodyObject where GameActorObject extends Actor and GameBodyObject extends GameActorObject respectively. In this section we will discuss and define actors and bodies. First we

digress to a discussion of some OGRE classes, namely the scene node and entity classes. These classes are integral to both the GameActorObject and GameBodyObject classes.

3.6.2.1 OGRE Scene Nodes and Entities

OGRE uses a scene graph implementation to manage the transformations (position, orientation and scale) of all objects that exist in the virtual world [32]. Nodes in the scene graph are called scene nodes. Each scene node can have many children, where each child can be either another scene node or an entity. Scene nodes also have an associated transformation that is applied to every child, albeit a scene node or an entity. We formally define an OGRE scene node in definition 3.6.1 [32].

Definition 3.6.1 (OGRE Scene Node) *An OGRE scene node is a node in the OGRE scene graph that has the following properties:*

- *it can have any number of children,*
- *its children can be scene nodes or entities,*
- *it has an associated transformation, and*
- *the associated transformation is applied to all of its children.*

OGRE entities form the basis for all renderable objects in the virtual world. Entities can have meshes associated with them and skeletons for meshes that have skeletal animations. Entities also have an associated local transformation. Thus the avatar of a player will be represented by an entity. This entity will be added to a scene node, and as the player issues commands to move the avatar, the transformations on the scene node will be modified in order to move the avatar. If the avatar has animations, these will be played through the entity class. We formally define an OGRE entity in definition 3.6.2 [32].

Definition 3.6.2 (OGRE Entity) *An OGRE entity forms the basis for all renderable objects in the virtual world. OGRE entities can be placed in the OGRE scene graph as children of scene nodes. An OGRE entity has the following properties:*

- *it can have an associated mesh,*
- *it can have an associated skeleton, if the mesh is skeletally animated, and*
- *it has a local transformation.*

At this point we have defined the necessary background to be able to continue our discussion on actors and bodies.

3.6.2.2 Actors and Bodies continued

Actors in the virtual world can be described as those objects that have at least the following properties:

- a position,
- an orientation,
- a velocity,
- a shape, and
- a mass.

Actors do not, however, have any visualisation associated with them: one cannot see an actor on its own. A body is simply an extension of an actor where there is also some visualisation associated with it. Thus a body is:

- an actor with all of its properties, and
- an associated visualisation.

In the hierarchy shown in Figure 3.3 on page 43, the Actor class encapsulates the actor implementation in the underlying Nvidia PhysX physics and collision detection system. We extend the Actor class with GameObject to yield the GameActorObject class. The GameBodyObject class then encapsulates a scene node and entity from OGRE to allow for visualisation of the object and extends the GameActorObject.

In this section we gave an overview on the design of the hierarchy of objects that forms the foundation for any visible or interactive entity in the virtual world. Throughout the design of the members of this hierarchy, we endeavoured to enforce extendability of these GameObjects as far as possible. In order to support physics and collision detection, the concept of an actor and the GameActorObject was introduced. Finally, in order to support visualisation we introduced the body concept with the GameBodyObject class that encapsulates an OGRE scene node and an OGRE entity.

We have discussed the structure and the members for the GameObject hierarchy. So far we have only discussed the structure of game objects in our engine without considering

their management. Therefore in the next section we discuss the management of game objects in our engine. Management includes creation, destruction, keeping track of and finding game objects that are in existence.

3.7 Management of GameObjects

In the previous section we discussed the structure of the GameObject hierarchy, focussing in particular on the GameObject, GameActorObject and GameBodyObject classes. Thus far we have not discussed the actual management of GameObject instances created during run-time.

During execution of a game or therapy tool that uses our engine, thousands of instances of game objects could be created and destroyed, depending on the requirements of the game or therapy tool in question. Certain systems in the engine might also need to be able to find game object instances by their names. Preferably, some central class must be used to create instances of objects from the hierarchy. The type of object instantiated can be based on parameters passed to a function in the class, and a pointer to a base class instance from the hierarchy (GameObject) can always be returned. Such a scheme should assist in ensuring that changes made to the hierarchy remain invisible to games or tools implemented with the engine. This is important, since plug-ins could add new classes to the hierarchy. Using such a management class also makes it possible to provide functionality for finding instantiated game objects by their instance names. We summarise what we require from such a central management class in the following points:

- the class provides an interface for the creation of instances of objects from the game object hierarchy, but keeps the specifics hidden from users by only returning pointers to instances of a base class in the hierarchy,
- the class should be extendable to support the creation of new classes by plug-ins and
- the class can return pointers to instantiated classes by searching with their instance names.

In the next section we discuss the design of this central management class.

3.7.1 The Game Object Factory

An abstract factory is a class that is responsible for creating instances of classes from a class hierarchy with an abstract base class where the exact class instance to be created

is not directly specified. Abstract factories are discussed in more detail in Section A.2. Essentially abstract factories allow one to create instances of classes where the details are hidden since only pointers to the abstract base class is returned.

We therefore design a GameObjectFactory class, that is an abstract factory, for the creation of class instances from the GameObject hierarchy. Our motivation for using an abstract factory to create instances from the GameObject hierarchy is that it keeps the specifics behind creating instances hidden from users, by only returning pointers to instances of a base class in the hierarchy. Also, it can be extended to create instances of new classes that could be added to the GameObject hierarchy through plug-ins. A number of creation functions are provided in the GameObjectFactory interface. These functions create various kinds of GameObjects depending on which function is used and what parameters are passed to it. The GameObjectFactory is also implemented as a singleton, ensuring that we have a single factory that creates GameObjects that is also easily accessible throughout the engine.

One feature that we still require, that is not part of the abstract factory pattern, is the ability to use the factory to find existing instances by their names. In the following section we discuss a slight variation on the standard abstract factory: namely factories that create instances derived from the NamedInstance and NamedObject classes discussed in Section 3.6.1.1. These factories provide special functionality that depends upon the fact that the instances that are managed have instance names and class names.

3.7.2 Named Object Factories

In the preceding sections we discussed the GameObject hierarchy. We discussed how GameObject classes have instance names and class names. We also presented the abstract factory design pattern as a method to use for managing the creation and destruction of instances from the GameObject hierarchy. In this section we go one step further to include the instance and class naming functionality with the abstract factory design pattern to create factories that can create, destroy and find instances by using their names.

The NamedObjectFactory class provides an interface which can be used to implement object factories that create instances that are derived from the NamedInstance class. Such a factory is fully aware of the fact that the instances that it creates have names, in fact, the factories go further and enforce the condition that names of these created instances are unique, at least within the scope managed by the factory. The NamedObjectFactory interface also defines methods, using the unique names of instances, for checking whether an instance exists, getting a reference to an existing instance and destroying instances.

Other than being aware of instance names, these factories are also aware of class names. NamedObjectFactories support the creation of instances of a certain class by specifying only the class name and instance name as parameters. This functionality becomes useful in the design of the serialisation and networking systems in the engine.

In this section we discussed the use of object factories in the engine. We discussed the creation of instances from factories where the class name and instance name of an instance to create are the only specified parameters. In the next section we discuss how the engine utilises OGRE to render a real-time 3D scene.

3.8 Rendering

We use OGRE as a major component in our engine and specifically for all graphics rendering. Therefore we need to design the necessary classes to allow our engine to manage OGRE and use it to render a scene when appropriate. In this section, we discuss how this should be accomplished.

The rendering of the virtual world requires a task, which we call the VideoTask (tasks were discussed in Section 3.2). When the VideoTask is initialised by the kernel, it should initialise OGRE and set the applicable configuration parameters for OGRE. The parameters include the screen resolution, whether full screen or windowed mode should be used and also which rendering system to use. OGRE supports multiple rendering systems through plug-ins. Our engine must detect whether the Microsoft® Windows platform is being used and then try to use Microsoft® DirectX for rendering. In the case of other platforms, the OpenGL rendering plug-in for OGRE should be used. When the engine kernel updates the VideoTask, the task must use OGRE to render the current scene.

In Section 3.5 we discussed the physics and collision system that we use in the engine. We use Nvidia PhysX along with the NxOGRE add-on for OGRE. There is one issue that is problematic when NxOGRE and OGRE are used. OGRE and NxOGRE both have their own classes for scene management: OGRE has a SceneManager class and NxOGRE has a Scene class. In order to provide one logical construct for developers to access a scene in our engine, we must provide a Scene class specific to our engine. In order to address this situation we design a Scene class for our engine that wraps the use of the OGRE SceneManager and the NxOGRE Scene classes. The class provides the majority of the functions available in the two classes.

In the next section we discuss the design of the generic input handling framework of the engine.

3.9 Input Handling

Interactive 3D virtual environments must provide support for user input. A user must be able to control her avatar in the virtual world through some form of input mechanism. In this section, we discuss the design of the input handling framework in our engine. Furthermore, we discuss the aspects of the design that lend extendability to our framework for the support of non-standard input devices in the future. Since our engine is to be used to create educational games and ASD therapy tools, and individuals with ASD often suffer from motor difficulties [26, 27], we have to ensure that in future support could be added to our engine for non-standard input mechanisms, such as speech recognition. To accomplish this, we design a generic framework for input devices. In order to add support for a new input device, certain base classes should be extended to implement the specific functionality. These extended classes can then be used transparently in the engine. In this section we discuss the design of the generic input framework.

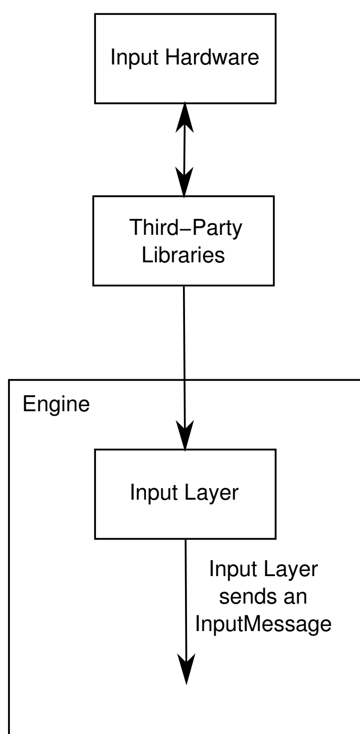


Figure 3.4: An overview of the hardware abstraction layer. Third-party libraries manage input hardware and communicate events and state changes to the input handling layer in the engine. This layer then creates InputMessages that encapsulate these input events.

There are two common methods for handling input [51]. The first is to periodically poll the state of a device. This entails, for example, periodically getting the state of the keyboard and checking whether any keys are pressed or not, or whether there are any changes between the current state of the device and its previous state. The second

manner in which input is handled is through an event system. In an event system the state of a device is not directly polled. Instead an event would be fired when the state of a device changes. Such an event would usually encapsulate the specific change. In general, direct polling of device states is used in lower level libraries that can access input devices directly. Event systems also tend to be somewhat easier to work with. One simply has to define a function to be called in case a specific input event occurs. This means that programmers that utilise an event system can be less concerned with hardware specifics and focus on responding to specific input events. In our engine we require an abstract input handling layer so that there is a well defined way in which new input devices can be added to the engine. A broad overview of our input system is given in Figure 3.4. Our input handling layer uses the message-passing system of the engine to communicate input events to developers utilising our engine. This does not mean that device state polling is never used, however. Implementations for specific hardware can still use device state polling at a lower level, as long as these implementations ultimately create messages encapsulating the event and use the input layer defined in our engine. Therefore, the goal of our input handling framework is to create a hardware abstraction layer, which separates implementations for specific hardware from the manner in which the engine reports input events. We now discuss our input handling framework in more detail.

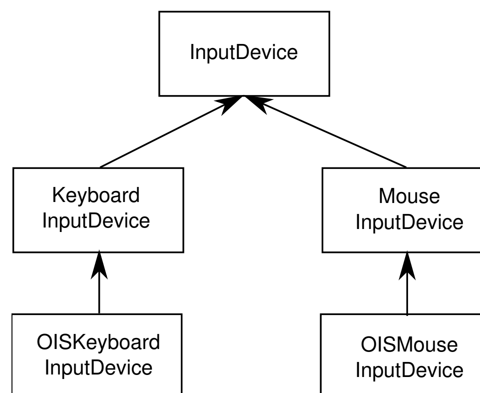


Figure 3.5: The hierarchy of InputDevice classes.

The hierarchy of classes used to represent input devices in the hardware abstraction layer in our input framework can be seen in Figure 3.5. An abstract class, InputDevice, forms the base class for any type of input device that could be supported by the engine. Specific input device implementations should extend this base class. Standard keyboard, mouse and game pad support is implemented through the Object-oriented Input System (OIS) [12] open-source library. The OIS library provides an object-oriented API for cross-platform support of standard input devices. The basic functionality of our engine should include support for such standard input devices. We accomplish this by providing extensions of the InputDevice class for generic keyboards and mice in the KeyboardInputDevice and

MouseInputDevice classes. These classes are then specialised for OIS in the OISKeyboard-InputDevice and OISMouseInputDevice classes. When the OIS version of the classes are instantiated, the classes would be configured to communicate with their counterparts in the OIS library. Our input system would need to frequently check whether input events have occurred. We accomplish this by including an input task. When the kernel updates the input task, the task updates the OIS library. OIS allows one to use polling or buffered (event driven) input handling. We shall use buffered input with OIS. Thus if input events have occurred, the OIS library would call the appropriate functions in the OISKeyboardInputDevice and OISMouseInputDevice instances that would send messages to the input task that describe the input events. Since messages must be sent with the message-passing system and these messages are specific to input events, we would need to create an InputMessage class for this communication. In this design we again have a hierarchy of classes with an abstract base where plug-ins can define new classes but we only use a reference to an instance of the base class throughout the engine. Therefore InputDevice instances are created through a NamedObjectFactory implementation, namely, the InputDeviceFactory.

We now discuss the input event life-cycle, as shown in Figure 3.6. When an input event occurs, the input task must receive an InputMessage from one of the underlying InputDevice classes. The message should then be analysed, upon which a function callback would possibly be called, and the message should then be sent to any other instances that have been registered to receive InputMessages. The callback that could be called when an InputMessage is received, is based on an action callback system. Actions are defined as simple strings and are mapped to input device events. Each input device can define a number of string based events that describe the input event that occurred. For example, the OISKeyboardInputDevice generates a string “keyboard0.A” when the “A” key is pressed on the keyboard. The associated InputMessage contains more information about the event, such as whether the key was pressed or released. The input task implementation uses the configuration system of the engine to determine to which action string, if any, the event string is mapped. Once the action string is known, the input task uses the InputActionManager to check if there is a callback associated with this action. If a callback is found, it is executed.

Our motivation for including an action callback system is largely to facilitate ease-of-use. Developers of games and therapy tools now have a choice of whether to simply subscribe to receive InputMessage instances or to define specific events on a device that should call specific functions. The functions that could be called could be defined in scripting languages and are then called through functors. Scripting languages and functors are discussed in more detail in Chapter 4.

In this section we discussed the design of the input system of the engine. In the following

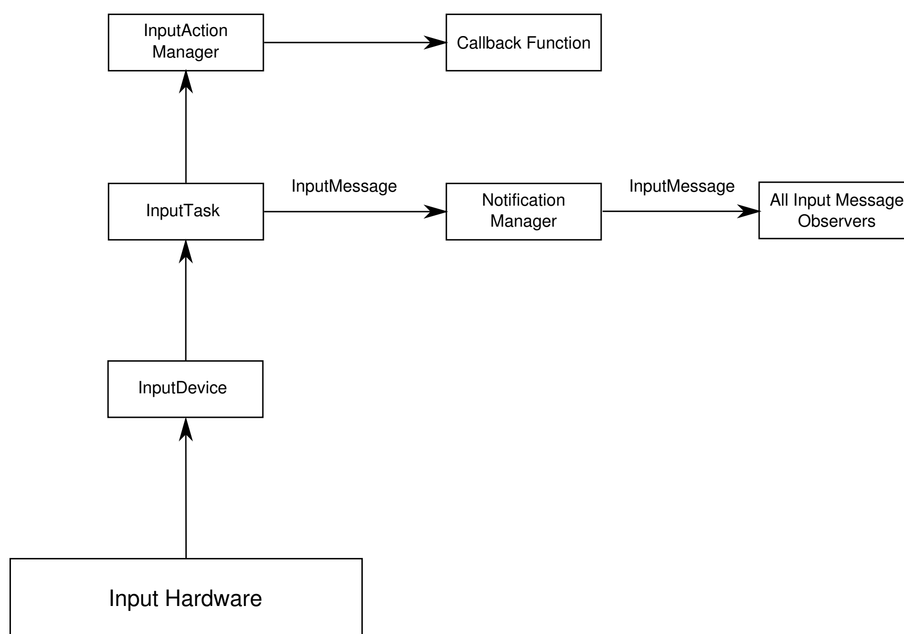


Figure 3.6: Handling an input event.

section we discuss the design of the audio framework of the engine.

3.10 Audio

Audio is a standard form of output in 3D virtual environments. In our case audio support is important since it allows us to reach younger individuals that can not yet read at a satisfactory level. In this section, we discuss the design of the audio framework in our engine.

In order to satisfy the extendability requirement of the engine, we design a generic audio framework. The framework is shown in Figure 3.7. This framework must be extended and implemented by audio plug-ins to provide specific audio functionality to the engine. We define a class hierarchy with an abstract base class, similar to the design of the task hierarchy and the game object hierarchy. The base class of this hierarchy is the Sound class. The interface of this class includes functions to play, pause and stop playback of the sound represented by an instance of the class.

The size of audio files can differ dramatically. Background music or speech audio files are typically large. Other sounds, such as foot steps, are usually short and small clips but would be played back often in quick succession. It makes sense to distinguish between these types of sounds. We therefore define two new sound class types in the sound class hierarchy, namely the StaticSound and StreamSound classes. Static sounds are those sounds that are in small files and are played repeatedly in short succession. Static sounds

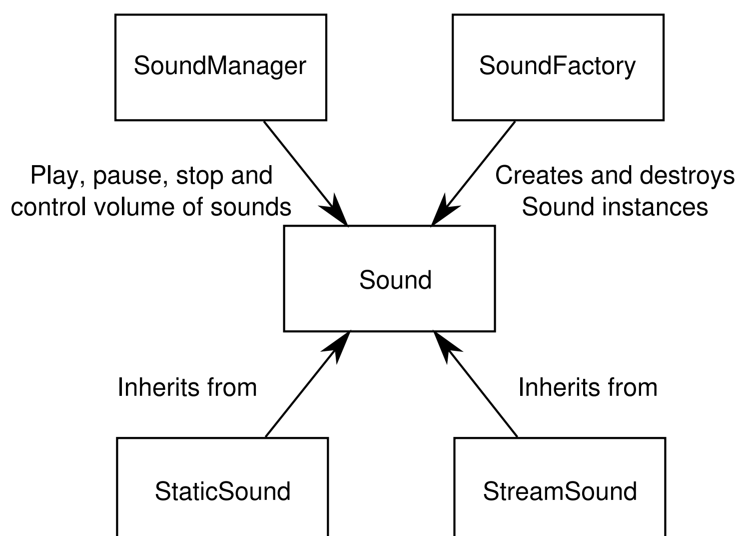


Figure 3.7: An overview of the design of the audio framework.

would be completely loaded into primary memory. Stream sounds are used for larger sound files, such as background music, where the entire file is not necessarily loaded into memory. During playback the file is streamed from secondary storage.

Since we have a class hierarchy with an abstract base class, and plug-ins must be able to define new sound types that extend existing types in the hierarchy, we use an abstract factory (see Section A.2) to manage the creation instances of sound classes. This again allows us, as with the game object hierarchy in Section 3.6, to use the abstract base class (in this case the Sound class) in the engine API, but through polymorphism developers can use any classes that inherit from the Sound class. In our design we call this abstract factory the SoundFactory. Our engine favours an event driven design, where message-passing and function callbacks (see the input handling framework in Section 3.9), is used for communicating events. Therefore, it makes sense that we should support a manner in which notifications can be sent when a sound starts and stops playing. Furthermore, it would also make sense to provide some central point from which it would be possible to pause or stop all audio playback that is occurring at a certain point in time. Therefore, we include a SoundManager in our design. The SoundManager provides virtual functions for stopping all audio playback and for managing a callback system where users can specify functions that should be called when a specific sound starts or stops playing.

During the design of this audio framework we investigated the OgreOggSound audio plug-in for OGRE. Much of our design is based on concepts found in the design of OgreOggSound, since it includes a static sound, stream sound, sound factory and sound manager in its design. In Section 4.4 we discuss an implementation of an audio plug-in for our engine that uses the OgreOggSound plug-in.

In the next section we discuss graphical user interfaces (GUIs) in our engine.

3.11 Graphical User Interfaces

The engine we are designing will be used to create educational games and ASD therapy tools. Most games require some form of GUI, even if it is only used for configuration purposes or entering the name of the player. Educational games might require a more robust GUI. Certain educational games might present the player with a number of exercises, and the player would need to use a GUI to provide the answers. Therefore our engine requires a GUI system. Furthermore, individuals with ASD can be easily distracted by ill-conceived GUIs [36] and therefore we must ensure that our GUI system is customisable. In order to improve the ease of use and content creation pipeline of the engine, we also need a tool that can be used to create GUI layouts. Therefore we require the following from a GUI system for our engine:

- the GUI system must support OGRE, since we use OGRE for rendering purposes,
- the GUI system must have a layout editor that allows for easy creation of GUI layouts, and
- the appearance of the GUI elements must be modifiable.

There are two GUI add-ons available for OGRE, namely, CEGUI [63] and MyGUI [21]. Both have similar functionality and both have layout editors. Also, the appearance of the graphical elements of both CEGUI and MyGUI can be modified relatively easily. We tested both add-ons, and found that certain aspects of CEGUI were somewhat difficult to use. Specifically, we encountered difficulties with the drag and drop functionality. MyGUI is a relatively new library with a simple API. We therefore use MyGUI in our engine, primarily since it is a smaller library and certain aspects of it are easier to use than CEGUI.

An overview of the design of the GUI system in the engine is given in Figure 3.8. In order to integrate a GUI system with our engine design we must determine what the GUI system would require from the engine. Users must be able to interact with the elements of a GUI system that are visible to them. This implies that the GUI system must be able to receive information from the engine regarding input. In Section 3.9 we discussed the input handling framework of the engine. The GUI system should be notified of all input events, since there is no way for the engine to know what would be valid input for the GUI or not. Thus the GUI system must receive InputMessage instances that are sent by the input framework. The GUI library must then be given the opportunity to process these messages in between frames rendered by the VideoTask, since input can obviously change the appearance of the GUI. Therefore any GUI system that is used with our engine must have a task that subscribes to input messages and converts these

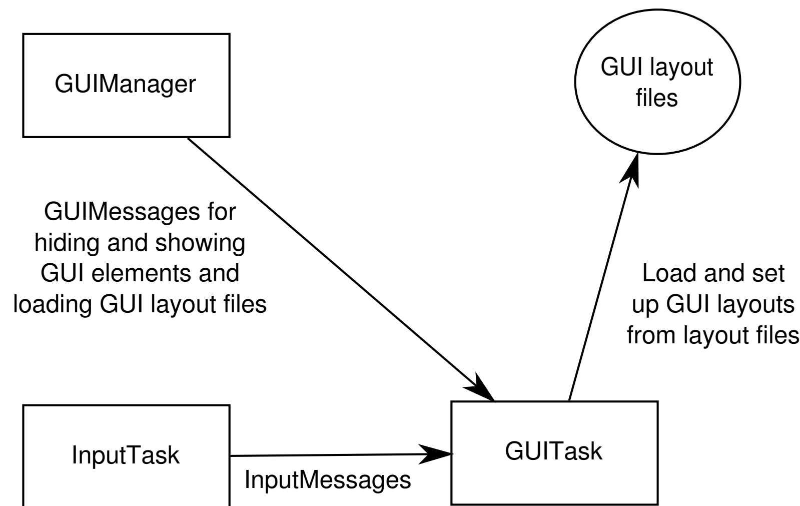


Figure 3.8: An overview of the design of the GUI system in the engine.

messages into valid input for the particular GUI library. Thus our design must include a GUITask. Furthermore, developers must be able to control certain aspects of a GUI in their games or therapy tools. Consider a menu system in a game, for example. When the user clicks on a button, new windows and buttons might appear or disappear. Therefore developers need a way in which to control certain aspects of the GUI system so that proper logic behind GUI transitions can be implemented. This implies that there needs to be a class in our design that allows developers to control certain aspects of the GUI. Therefore we include a GUIManager class in our design (see the discussion on manager classes in Section 3.3). This manager class must allow developers to call functions to hide or show certain GUI elements, hide and show the GUI cursor, switch between GUI layouts and possibly even create and destroy GUI elements dynamically. The most important functions would be those of hiding and showing elements and switching between GUI layouts. For most games and therapy tools developers should create all the necessary GUI layouts with the MyGUI layout editor. Then, using the manager class, they can easily switch between layouts in the game or therapy tool. Communication between the GUIManager and GUITask must take place through message-passing. This implies that we also need a message class that is designed specifically for GUI messages. Therefore we include a GUIMessage class in our design. The message class is reasonably generic and contains no MyGUI specific functionality. It contains generic information about the hiding and showing of GUI elements, hiding and showing the cursor and switching between layouts.

An important consideration in the design of the GUI system for the engine is to what extent an abstract interface for GUI integration must be included. In our design the GUITask and GUIManager is implemented with specific functionality for MyGUI. Plug-ins for other GUI systems can easily be implemented by following a similar process and

providing their own tasks and manager classes. However, we provide no abstract hierarchy of classes for GUI elements, such as buttons and text boxes. The disadvantage of not including such a hierarchy is that there is not a generic way by which to directly access the class instances of GUI elements. If a game or therapy tool needs to access a button to change its caption, for example, then the developer needs to know what GUI system is being used and use a specific solution for that GUI system. Functionality to facilitate this could be included in the manager class. Hence, the disadvantage of this situation is that an existing game or therapy tool might need to be modified if the developer switches between GUI plug-ins. The reason for not including such a hierarchy is that GUI libraries can differ substantially and defining an abstract hierarchy that will work in general can be problematic. The advantage of not including such a hierarchy is that it makes it substantially simpler to implement new GUI plug-ins for the engine, since all that is needed is a task and a manager class specific to the GUI library. Lastly there are also some performance considerations. The base classes in such a class hierarchy would contain numerous virtual functions. Although virtual functions work well to provide abstraction through polymorphism, there is a performance cost associated with the process of finding the correct function to call at run-time.

In the next section we discuss the design of the data capturing framework in the engine.

3.12 Data Capturing

Therapists stressed the usefulness and importance of capturing the actions a user takes in the virtual world, as well as a full replay facility for any particular session. The replay system is discussed in Section 3.14. In this section we discuss the design of the data capturing system in the engine.

We need to design a generic data capturing system, as we have to cater for many different kinds of captured data. Developers of games and therapy tools must be free to capture almost any type of information that is applicable to their game or tool. An educational game, for example, could present a player with a number of exercises and request answers from the player. The data capturing system must be able to capture the exercises posed, the answers the player gave and the identity of the specific player. Any data that is captured would need to identify the user that was using the therapy tool when the data was captured, as many different individuals may utilise the same computer for playing games or using therapy tools. A logical conclusion, therefore, is that the data capturing system must make provision for storing a list of unique users. The data capturing system must also then be able to associate a user with any data that is captured. These requirements lead us to consider the standard relational database model where related data is stored in a relation,

typically called a table [49]. Thus there could be a table for storing player information and, in the case of the educational game example, a table for storing the answers that players provide, where each answer references the specific player that provided the answer. Thus we use a simplified relational model for the design of our data capturing framework.

	Attribute 0	**Attribute** 1	$\cdots$	**Attribute** n
Entry 0	Value 0	Value 1	$\cdots$	Value n
Entry 1	Value 0	Value 1	$\cdots$	Value n
$\vdots$				
Entry m	Value 0	Value 1	$\cdots$	Value n

Figure 3.9: An entity, as a table, with its entries. The underlined columns constitute the key for an entry.

In our design we define entities as tables or relations. This model is illustrated in Figure 3.9. We define the attributes of the entity as the column descriptions in the table. We define a row in the table as an entry in the entity. We also include the concept of keys, where if an attribute of an entity is defined as a key, each entry in the entity must have a unique value for that attribute. Therefore keys can be used to reference unique entries in an entity.

Since the basic design of the data capturing system follows a relational model and the use of SQL in relational databases is well known [49], using an SQL database as the storage medium for our data capturing seems natural. We design our data capturing system with a number of abstract base classes, where the functions responsible for reading and writing to the storage medium are not implemented. Thus it is possible to implement the data capturing system using many different database or storage types. Even if one uses an SQL database, there is also the issue of whether the database is local or remote. We therefore include provisions in our design for notification on the completion of read or write operations. Confirmation of the completion of read and write operations is important, since remote database implementations will likely utilise delayed read and write operations. Users of the data capturing system can use this notification mechanism, even if it is known that the storage medium used is local, so that it would be possible to potentially switch to implementations that use remote storage as well.

An overview of the design of the data capturing framework is given in Figure 3.10. In our design, entities are represented by instances of the GameLogEntity class and entries are represented by instances of the GameLogEntry class. Since entries are sets of values associated with an entity, the GameLogEntity class manages and creates GameLogEntry instances. We also require a class that communicates with the underlying storage medium and manages entities. We therefore include the GameLogManager class in our design. The GameLogManager class has virtual functions for reading entities from secondary storage

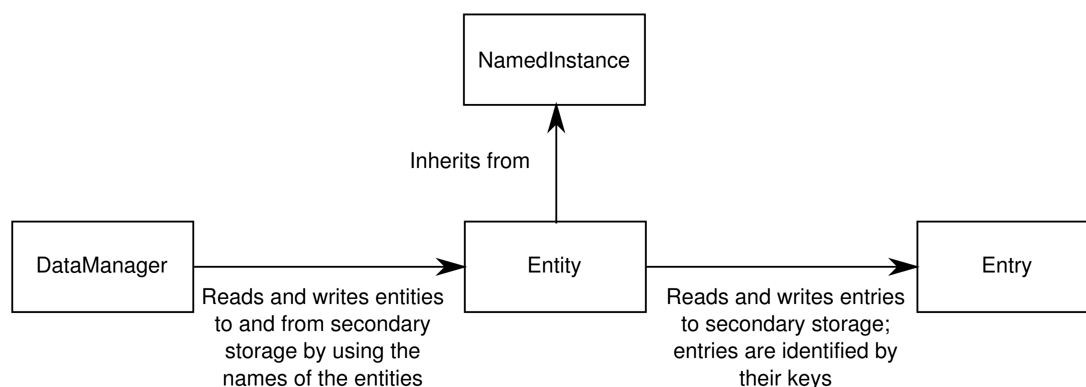


Figure 3.10: An overview of the design of the data capturing framework.

and writing entities to secondary storage. These virtual functions must be implemented for specific storage systems, such as an SQL database. We discuss such an implementation in Section 4.5. In our design the manager class is used to read entities into memory and to write modified entities to secondary storage. Our system does not automatically synchronise entities in memory with the versions on secondary storage. This provides developers of games and therapy tools with as much control as possible over what is ultimately saved by the data capturing system. Furthermore, the manager class has a number of functions that can be used to check whether read and write operations have completed.

In this section we discussed the design of the data capturing system. It follows a relational model and is thus well suited for SQL database implementations. We provide a manager class with virtual functions for read and write operations that must be implemented by specific database or storage implementations. Furthermore, the manager class also provides functionality for checking whether read or write operations have completed, since certain storage implementations might utilise delayed reading and writing. In the next section we discuss the design serialisation framework in the engine.

3.13 Serialisation

In this section we discuss the design of the serialisation mechanisms in the engine. Serialisation is the process whereby we convert information in our engine into a byte-stream that can be saved to a file or transmitted across the network. Serialisation forms an important part of the design of the replay (discussed in Section 3.14) and networking (discussed in Section 3.15) systems in the engine.

The serialisation mechanism must be able to serialise certain class instances so that these instances can be transmitted over a network or saved to a file. We cannot simply provide

a central class with functions for serialising different types of classes, since this would require one to add new functions to this class for every new class type that is added to the engine. A better, more generic and extendable approach is to provide an abstract base class, namely the Serialisable class. All classes that must be serialisable must inherit from this class. The Serialisable class then defines virtual functions (in our design these functions are called serialise and populateInstance) that when called would serialise the class or populate a class instance from a serialised version. We thus leave the details of how a class a serialised up to the specific class implementation. The Serialisable class must provide a number of implemented functions for serialising primitive types, such as integers, floating point values and strings. If a class encapsulates other class instances, those encapsulated classes should be serialisable as well. This would allow a class to call the appropriate serialising functions for the classes that it encapsulates.

A possible improvement on this design is to provide a class hierarchy of serialiser classes. Serialiser classes contain the functions for writing and reading primitive types to some form of storage, such as a memory buffer or a file. In this way different serialiser implementations could be created for different needs. Some serialiser could simply create byte-streams that are saved in memory. Others could use file streams or compressed file streams. Furthermore, these implementations could ensure that the order in which the bits of a byte is stored in memory is operating system independent, ensuring true cross-platform serialisation capabilities. Such a hierarchy of serialiser classes are present in the Enginuity Engine, but is not part of our design at this point in time, due to time constraints.

In the next section we discuss the design of the replay system in the engine.

3.14 Replays

Therapists indicated that a replay system in our engine would be a useful feature. A replay would be some form of recording of the state of the virtual world when a user is playing a game or utilising a therapy tool. Such a replay would allow therapists and teachers to review a particular user session and to potentially see certain behaviours that might be difficult to capture in the data capturing system. Some methods for recording replays are discussed in [18]. There are three methods that come to mind. The first is to simply save a video, in some form or another, of the session. The second would be to save the state of the virtual world at certain key intervals. These states could then be played back and some interpolation techniques could be used in order to move from one state to the next. The third possibility relies on the assumption that the evolution of the virtual world would be deterministic given all of the input that it receives. Thus if we record the

start state of the virtual world and then record all of the input that it receives, we should be able to recreate a session by restoring the start state of the virtual world and playing back the recorded input.

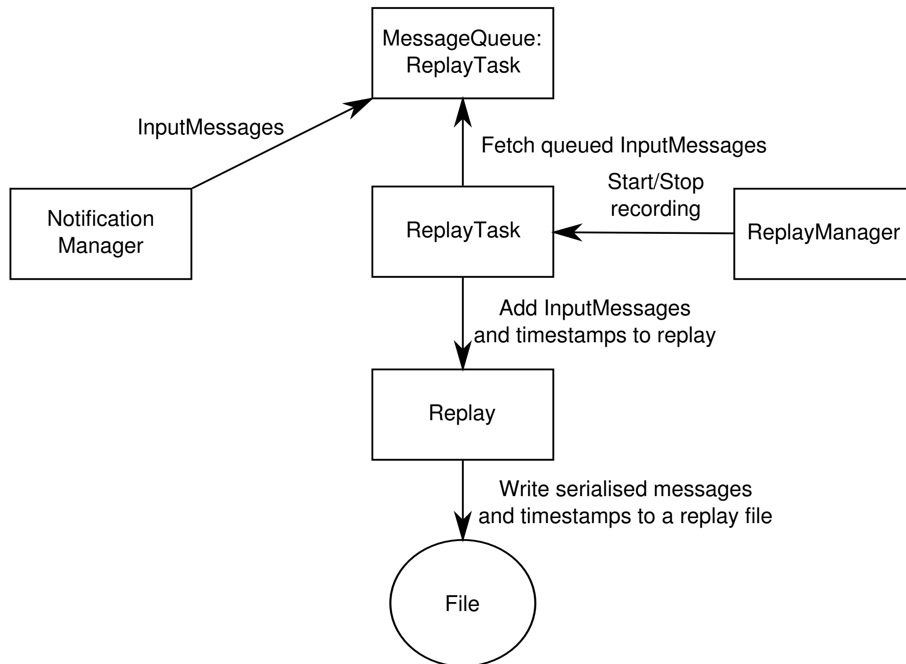
The disadvantage of the first method – saving a video – is that it is prohibitively expensive in terms of computation and storage. Saving a video, without separate video capturing hardware, while the 3D virtual world is active, would negatively impact the performance of the simulation and would detract from the user experience. The advantage of the video capturing method is that it would arguably be the simplest method to implement. The disadvantage of the second method – saving the state of the virtual world at key intervals – is that it requires complicated interpolation techniques to move from one saved state to another. The advantage of the method is that, if it is implemented correctly, it should work on any system and there should be no way in which a completely incorrect playback of the replay could occur. The disadvantage of the third method – saving all of the input that the engine receives – is that there would be no guarantee that playback always occurs correctly. The method is highly temporally dependent and relies on our engine being deterministic. All input events must be played back in the correct order with the correct time between events. If a replay is played back on a system that encounters performance issues, it is possible that the time between the recreation of certain input events would be larger than it should be. This would cause the replay to play back incorrectly. We can alleviate this problem somewhat by establishing a minimum frame rate at which the replay playback will be correct, by lowering the execution intervals of the input and video tasks when recording a replay. This effectively lowers the rate at which input can be received, increasing the time between messages about input. We can potentially address the synchronisation issue by also periodically including the state of the world in the replay. This state of the virtual world would then periodically ensure that the replay is playing back correctly. One advantage of using the input event recording method is that it integrates well with the design of our engine: all input events in our engine are encapsulated in InputMessages. Thus the replay system would easily be able to subscribe to these messages to save a replay. Furthermore, this method would remain valid, even if the engine is extended, as long as input events are encapsulated in InputMessage instances or in class instances that are derived from the InputMessage class. If extensions to the engine change the game object hierarchy, it would be possible that changes would be introduced to the manner in which the state of the virtual world would be saved. Thus such changes could necessitate modifications to the second replay method. A further advantage to saving input events is that it would also automatically make it possible to save any interaction a player has with a GUI. This can be especially useful for therapists, in our context. It would allow them to see how children interact with the GUI and how they move the mouse around.

Therefore, we use the method of saving all input events in the design of our replay system. Our primary motivations for using this method are:

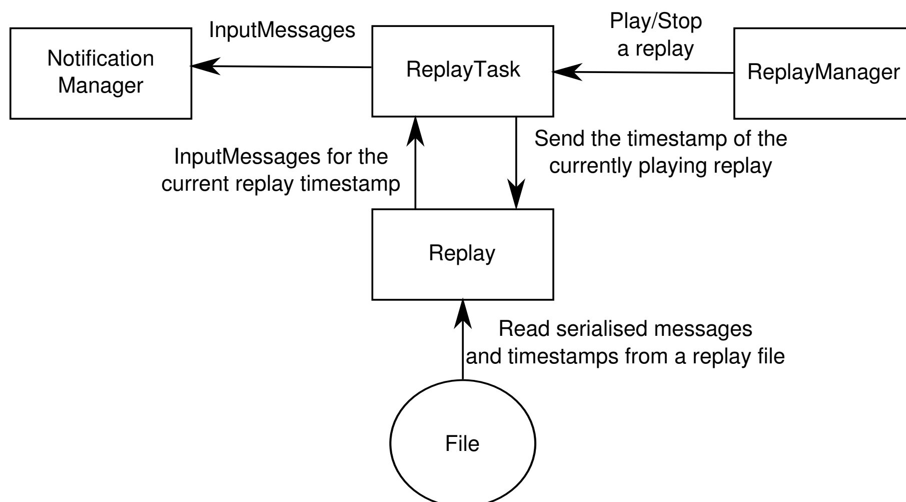
- the nature of the method utilises the communication system of the engine,
- since all input events should be encapsulated in InputMessage instances, this method should remain valid even if the engine is modified in the future,
- it is simpler to implement than saving the state of the virtual world and then interpolating between states, and
- it automatically supports capturing user interaction with a GUI and exact mouse trails.

The input event capture method requires that our engine and implemented games and therapy tools are deterministic. In order to make our engine deterministic we must consider a number of factors. Firstly, subsystems in the engine should not be dependant on the number of times it is updated but on the time between updates. This is provided through our task management interface. Secondly, we include a basic random number generator implementation in our engine for which we can save and restore the seed. Lastly, we must ensure that the third-party libraries we use are deterministic. According to the documentation of Nvidia PhysX, the physics system is deterministic if fixed time steps are used and if the physics solver uses a single thread [40]. OGRE is deterministic since its rendering process does not modify the state of the virtual world. Thus, we can ensure that our engine is deterministic.

The design of our replay model for recording replays is shown in Figure 3.11(a). The design for playing replays is given in Figure 3.11(b). We now discuss these designs in more detail. When recording a replay, our replay system must receive InputMessage instances. It must also keep track of the amount of time that passes between InputMessages that it receives. When our system is playing a replay, it must send the InputMessage instances that it previously recorded. It must also ensure that it sends an InputMessage instance after the correct amount of time has passed since the start of the replay playback. Thus our design should use a task that is frequently updated by the kernel. The task can register to receive all InputMessage instances that are sent in the engine, for recording a replay. Furthermore, a task would receive the time since its previous update every time it is updated by the kernel. Therefore, the task would have the timing information that our replay model requires. When recording or playing back a replay, the task could accumulate the total time that has passed since the beginning of the recording or playback process. It can then use this time to include a time stamp for an InputMessage when it is saved, or to determine the correct time at which to send the next InputMessage when a replay is being played.



(a) The design of the replay model for recording a replay.



(b) The design of the replay model for playing a replay.

Figure 3.11: The design of the replay models for recording and playback.

We include a Replay class in our design. An instance of this class represents a specific replay. The ReplayTask would be assigned such a Replay instance to use during recording or play back. When recording it should pass any received InputMessage instances, along with their time stamps, to the Replay instance. The Replay instance then serialises the InputMessage instances and their time stamps and saves it to a replay file. When a replay is being played back, the ReplayTask should call a function in the Replay instance with the current playback time stamp of the replay. At the correct intervals the Replay instance then returns the applicable InputMessage instances, loaded from the replay file. The ReplayTask should then send these messages through the engine, essentially faking the input that was previously recorded. In order to manage the ReplayTask and Replay instances, our design also includes a ReplayManager class. This manager class has functions for recording, playing and stopping replays.

When a replay is recorded, the state of the random number generator is saved. When a replay is played back, this state is restored. This helps to ensure that the virtual world simulation will be deterministic. We must also consider the state of the virtual world when replay recording begins. We have to serialise the state of the virtual world at the beginning of the recording process and recover this state when the replay is played back. Currently, the design of our replay system does not include serialising this initial state, due to time constraints. Therefore, it would be best to start recording a replay at the beginning of a level or scene in a game. When a replay is played back, the specific scene in which the replay was recorded should first be loaded and the replay must be played back from the beginning.

In this section we discussed the replay system in the engine. We use input event recording and rely on the determinism of the engine. The primary advantage of this design is the fact that it will remain valid as long as InputMessages encapsulate all input that a user can give to the system. The design should remain sound for various game or tool implementations and should withstand additions or modifications to the engine. The disadvantage of this design is that playback of replays will not be exactly correct on systems that have excessively low frame rates or considerable starvation of the thread in which the replay is played back. For example, if a message must be sent to stop an avatar that is moving in a certain direction and that message is not sent at the time specified in the replay, the avatar could move a larger distance than it should. This results in the incorrect playback of the replay. This is not a major concern since replays will likely be played back on the systems used to capture it or on systems with a similar hardware profile. The problem could be addressed by periodically saving a snapshot of the virtual world, with the current positions and properties of all movable objects in the virtual world, to the replay file. If synchronisation issues occur during playback of the replay, these snapshots can be used to restore synchronisation. At this stage the design of our replay system does not include

this synchronisation functionality.

In the next section we discuss the design of the network components in the engine.

3.15 The Networking API

The therapy tools that we shall typically create will be single-user applications, with networking not an issue. However, therapists indicated that optional remote monitoring and control over the games and tools would be a desirable feature. For example, in a therapy session, a therapist might wish to pause a tool at a certain point to discuss the choices a user has. Or, a therapist might wish to change the game logic or increase the difficulty level, depending on the actions of the user up to that point. The requirements of the network component in our engine is therefore to allow at least one remote user to connect and perform some administrative actions. Furthermore, the network component should also be designed such that future extensions are possible.

Smed *et al.* [52] conducted a review on proven techniques for providing network functionality in networked virtual environments and multi-player computer games. The client and server model is a standard approach. One system is identified as the server and the rest of the systems are clients. Communication takes place between a client and the server only, and clients do not communicate directly. Generally the server would maintain the state of the game world and communicate any relevant changes or updates to the connected clients. If the networked session is not taking place on a high bandwidth, low latency network synchronisation issues can occur. Update messages to and from the server might not arrive in a timely fashion. In order for clients to not simply stop activity in the game world due to network delays, an extrapolation method is used to ensure smooth movement. These methods are generally referred to as dead reckoning. With dead reckoning, the future position of an object is determined by considering its current and previous positions. Typically the velocity and acceleration of the object at these positions are also used. From this information a projected future position is calculated. While the client is waiting to receive an update message about the position and velocity of the object, it uses the information from the dead reckoning technique to move the object. When an update message finally arrives, it is quite likely that the current position of the object, as calculated by dead reckoning, is not accurate. In such a situation a second phase of dead reckoning is used, called convergence. The dead reckoning method now takes into account the correct current position of the object as received through the network message. Using the last couple of points from the projected path of the object, along with the new accurate position received through the network, the method fits a curve to these positions. The object is then moved along this curve to converge with its correct position.

A second common network model is the peer-to-peer model [52, 67]. In this model there is no server; all connected systems are equals and all systems have their own copy of the state of the virtual world. Any actions that occur locally at a system is transmitted to all the other connected peers. Such a model works best in a high bandwidth, low latency network. It is also important that all network messages are received in the correct order by the peers. However, synchronisation issues can still occur. In such cases the peer-to-peer model keeps all peers synchronised by periodically sending a state of the virtual world, that is considered to be accurate, to all peers. Furthermore, dead reckoning is also typically used in order to ensure smooth movement of objects.

In our case, games and therapy tools will always be used on reliable local area networks in a classroom or office. Furthermore, we strive to make it as simple as possible for developers to use our engine to develop games and therapy tools. Hence, we use the peer-to-peer network model in our design. One of the major reasons for using this model is to make it as simple as possible for game and tool implementers to include network functionality in their implementations. In our design, remote players are added to the game as if they are local players with their own keyboards or mice. Game and tool implementers only have to, upon the connection of a new user or disconnection of a user, ensure that its unique local data, such as a player avatar, is created or destroyed on the remote host. After the connection is established, the network components in the engine should take care of communication among peers. All relevant internal messages should be forwarded to all connected peers. The philosophy is to make all hosts believe that all of the remote users are actually local. Thus no implementations or extensions to the engine need to take potential network issues into account. Handling network issues, such as synchronisation, is left to the networking components. However, the design of our networking components does not include synchronisation among peers at this stage, due to time constraints.

We design the networking components in our engine to allow for changes through plug-ins. A plug-in can either completely replace the networking model in the engine or can modify its behaviour in some way. This makes it possible for developers to change our network implementation or to introduce completely new implementations in the future. In the following paragraphs we give a more in-depth discussion on our networking model as well as how the behaviour can be modified through plug-ins.

Any object that should be transmittable over the network must implement the Serialisable interface. Hence, all message classes that should be sent among hosts, as well as most of the classes in the GameObject hierarchy, need to implement the Serialisable interface.

An example of the functionality of our network model can be seen in Figure 3.12. The network component of the engine must periodically transmit messages over the network. Our design includes a NetworkMessage class. This class can encapsulate other Message instances. In order to transmit messages over the network our design also includes a task,

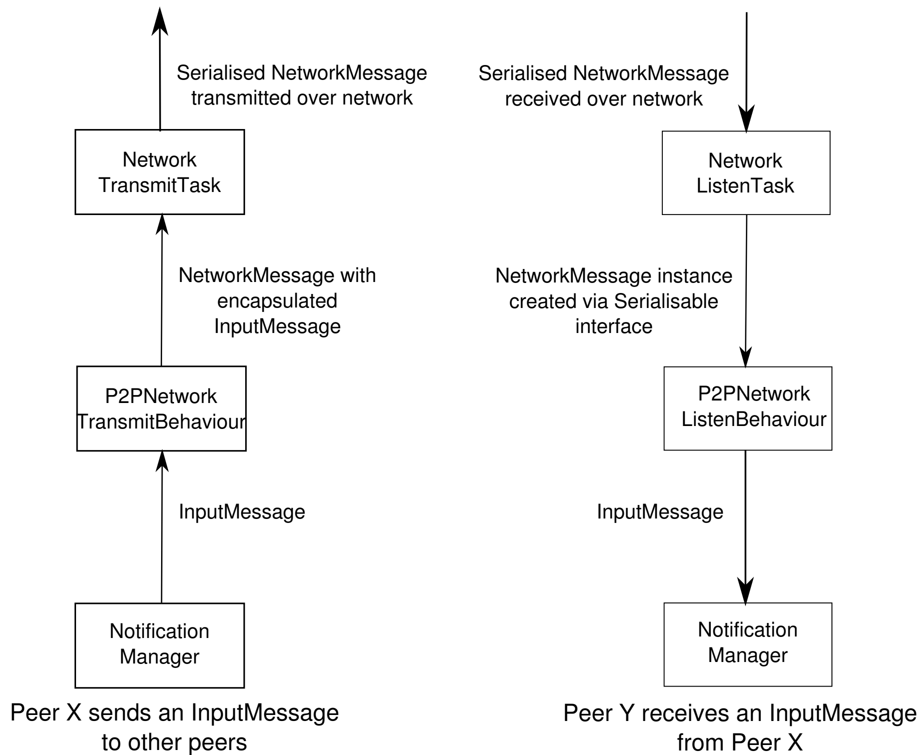


Figure 3.12: An example of the transmission of an InputMessage with the peer-to-peer network model.

namely the NetworkTransmitTask. This task receives NetworkMessage instances, serialises them and transmits them over the network. In order to receive messages we include a task, namely the NetworkListenTask. This task listens for incoming messages. When a message is received, the populateInstance function of the serialisable interface must be used to populate a NetworkMessage instance. This instance can then be sent throughout the engine.

We have not yet discussed the aspects included in our design to promote the extendability of our network system. We include the concept of so-called network behaviours in our design. Before a NetworkMessage is transmitted and just after it is received, it is sent to network behaviour classes. These classes can inspect the messages and take appropriate action, based on the network model that is represented by the behaviour classes. In our design we define a NetworkListenBehaviour class that acts as the abstract base class for all network behaviours related to incoming messages. We also define a NetworkTransmitBehaviour class that acts as the abstract base class for all network behaviours related to outgoing messages. The majority of the logic behind the peer-to-peer network model that we utilise in the engine is implemented in two classes, namely the P2PNetworkListenBehaviour class and the P2PNetworkTransmitBehaviour class. Finally, our design also includes a manager class, called the NetworkManager. The manager class is used to select which networking behaviour to use. This behaviour model allows plug-ins

to add new types of network models, or behaviours, to the engine.

Our peer-to-peer network model transmits messages about the creation or destruction of objects in the virtual world as well as certain relevant engine messages across a network to peers. Consider the example (see Figure 3.12, on page 66) of an InputMessage that is created when the user presses a button on the keyboard that should cause her avatar to move forward. The InputMessage for this input event must be sent to all connected peers in a network session. The P2PNetworkTransmitBehaviour registers itself to receive engine messages. It receives InputMessage created when the user pressed the button on the keyboard. The P2PNetworkTransmitBehaviour then encapsulates this InputMessage in a NetworkMessage instance and sends the NetworkMessage, along with the list of peers that should receive the message, to the NetworkTransmitTask. Using the Serialisable interface discussed in Section 3.13, the NetworkTransmitTask serialises the message and sends it to all the connected peers. The NetworkListenTask of a peer then receives this serialised message and creates a NetworkMessage instance from it using the Serialisable interface and sends the message to the P2PNetworkListenBehaviour. Finally the P2PNetworkListenBehaviour inspects the NetworkMessage, discovers the encapsulated InputMessage and sends it internally in the engine. Thus the input event is transferred from the host where it physically originated and duplicated on all connected peers. Similarly GameObject instances can be serialised and transferred to peers. This generally occurs if a new peer joins a network session and it must receive the state of the virtual world.

Since our peer-to-peer network model is primarily concerned with transferring engine messages among peers, and in particular InputMessages, it is similar to the replay system discussed in Section 3.14. Therefore, our network model also relies on the fact that the engine is deterministic with respect to the input that it receives. Our network model also supports the transfer of GUI input. This would make it possible for therapists to also remotely monitor the manner in which children interact with in-game GUIs.

Such a peer-to-peer model generally does not work for fast-paced games with a substantial number of players, since synchronisation problems can easily occur. In our case we shall generally have no more than two users in a session and the network infrastructure will be at least a 10Mbps reliable local area network. However, synchronisation issues will still occur, albeit much less pronounced than in the case of fast-paced games with many players. A future extension to this model would be to assign one of the peers as a quasi-host that periodically sends a more complete state to all other peers to ensure that peers remain synchronised. Pellegrino *et al.* [45] proposed such a modified peer-to-peer network implementation with a so-called central arbiter. This would enable the engine to easily handle a classroom-like situation, with ten to twenty users.

A final concern in the design of our network model is which underlying transmission protocol to use. The user datagram protocol (UDP) or the transmission control protocol

(TCP). UDP does not guarantee that a message will reach its destination, nor does it guarantee that a set of messages will be delivered in-order [47]. TCP is connection-oriented, reliable and ensures in-order message delivery. In our case the reliability of TCP is desirable since we do not include synchronisation, or mechanisms to ensure in-order message delivery, in the design of our network model at this time. However, UDP is commonly used in games since the reliability of TCP can cause higher latencies [67]. In our case low latency protocols are also important. Therefore, our design includes both UDP and TCP versions of the NetworkListenTask and NetworkTransmitTask classes, since both have desirable features for different situations.

In the next section we briefly discuss the design considerations around integrating support for scripting languages in our engine.

3.16 Scripting

In Section 3.1 we mentioned that modern game engines generally include support for scripting languages. This allows for a separation between the engine code and the game logic, since game logic is generally implemented in a scripting language and engine code is implemented in lower level languages. Furthermore, scripting languages allow for quicker testing of features since these languages are interpreted while the engine is running. In our case we envision the creation of future visual tools to assist in the rapid development of educational games and therapy tools. The output of these tools should be scripts that can used with our engine. Therefore, we include support for scripting languages in our engine design.

There are many existing scripting languages with C/C++ APIs. Two well-known languages used in game engines are Python and Lua [17]. Python is an object-oriented scripting language. Object-orientation in the scripting language can be important in our case, since our engine design relies on many principles from object-orientation. Lua is a more compact language and is often defined as a multi-paradigm language. It is primarily an imperative language without the concept of a class. It does, however, have a table data structure in which functions can be defined. In order to provide maximum flexibility, our engine design requires that support for scripting languages must be implemented through plug-ins. We discuss such implementation details in Section 4.7. However, it would be difficult for us to maintain a number of scripting languages simultaneously with the same level of quality. Therefore, we must select a primary scripting language for the engine and integrate it more closely with the engine design. We use Lua as this primary language for its relatively simplicity, compactness and speed. We discuss the implementation of the

scripting interface, both the primary Lua interface and the plug-in interface, in detail in Section 4.7.

In this chapter we discussed the requirements of our specific project, namely, to create a 3D virtual environment development platform, and elaborated these requirements into lower-level requirements similar to those of game engines. We discussed the design of our engine from the kernel and task management through to individual components such as data capturing, replays and networking. In the next chapter we discuss issues we encountered during the implementation of our design and how we solved these issues.

Chapter 4

Implementation

In this chapter we discuss various issues we encountered during the implementation of the engine and our solutions to these issues. We also discuss the implementation of various aspects of the engine that were only broadly discussed in Chapter 3. We discuss the difficulties encountered in implementing a multi-threaded, but thread-safe, kernel and task system. We also discuss the creation of an instance pooling system used to reduce the number of object instance creations and deletions. Also discussed in this chapter is the implementation of the generic input system of the engine as well as plug-ins for scripting language support. This chapter also includes a section that describes the language localisation system of the engine, allowing one to support text in multiple languages in developed games or tools.

In Chapter 3 it is mentioned that the engine supports multi-threading. In particular, the kernel (the kernel and task management system is discussed in Section 3.2) can manage tasks that are executed sequentially in the kernel thread, and tasks that execute in their own threads. In the following section we discuss some implementation issues regarding thread-safety and multi-threading in the engine.

4.1 Multi-Threading and Thread Safety

Since the engine has a multi-threaded design, it is important to ensure that all classes and functions that could be accessed by multiple threads at the same time, are indeed thread-safe[1]. In Section 3.2.1 we discussed thread-safety in the kernel. Specifically we mentioned the use of locks and the duplication of shared data as two ways to ensure thread-safety.

[1] When a class or function is thread-safe, precautions are taken to ensure that if multiple threads call the function, correct execution still occurs. This is generally achieved through some form of mutual exclusion where it is ensured that multiple threads cannot access a shared piece of data in a dangerous manner, such as one thread reading the data while another is writing to it.

The kernel duplicates its task list for thread-safety. In general, however, we use locks to ensure thread-safety throughout the engine. Specifically, we use the reader-writer lock implementation provided by POCO. The reader-writer locks are attractive since it provides a higher level of concurrency where many threads can read the same data at the same time, where with normal mutual exclusion locks we are always limited to only one thread being allowed to access shared data. However, the reader-writer locks of POCO do not support recursive write locks: the same thread cannot obtain multiple write locks on shared data, nor can it obtain a read lock on shared data if it already has a write lock on that shared data. That means that functions and APIs must be designed more carefully to eliminate such situations.

Therefore, our motivation for using the reader-writer locks and not the normal mutual exclusion locks stems from the fact that we can achieve a higher level of concurrent access to data with reader-writer locks than with normal mutual exclusion locks.

The use of multiple threads in our engine is complicated by the fact the some of the third party libraries we use are not thread-safe. These include MyGUI, NxOgre, OIS and to some extent OGRE. OGRE does have support for resource loading in a background thread, but the entire library is not yet thread-safe. In order to address this problem, we provide a lock for each library that is not thread-safe. This lock must be acquired before any function calls are made to the library.

In the following section we discuss some implementation issues we encountered with the message passing communication model and the timer task.

4.2 The Timer Task and Message Passing

In Section 3.2.1 we discussed the timer task and how it provides tasks in the engine with the amount of time that has passed since its last execution (we call this time between executions the time-delta). In the initial implementation of the timer task, this time-delta was delivered via the message passing system. When the kernel updated the timer task, the task would send a message that encapsulated the absolute running time of the engine and the amount of time that has passed since the previous update of the timer task (the time-delta). In Chapter 3 we discussed the temporal dependencies of tasks, and the fact that many tasks need this time-delta value. This requirement lead to the same timer task message being sent to a large number of tasks. This introduces unnecessary overhead, since a relatively large number of messages are sent whenever the timer task is updated. This lead us to test a system where only the kernel receives this timing information and then passes it as a parameter to each task it calls for execution. This change lead to an improvement in performance. This indicates that there is some overhead involved in the

process of creating and sending messages. Although, in this case, removing the message passing element and allowing the kernel to pass the time-delta along to tasks is a way of avoiding this overhead, in general this is not a favourable solution. The engine must have a robust, well performing communication system that can be further utilised by extensions and plug-ins in the future. Therefore, we have to investigate ways in which we can improve the performance of the message passing system in general.

Performance problems are introduced when numerous messages are sent frequently. A feasible explanation would be overhead introduced by the creation and destruction of numerous instances of message classes. Hence, we take two measures to reduce the number of message class instantiations. The first measure that we take is to ensure that when a message is sent, and it is to be received by multiple receivers, we never create clones or new instances of the specific message. We simply share the message instance among all receivers and only destroy the instance once all receivers have processed the message. This is accomplished via reference counting and is facilitated by the RefCountedObject and AutoPtr classes in the POCO library. The RefCountedObject class contains a reference count: an integer that is increased when the duplicate() function of the class is called and decreased when the release() function of the class is called. When the reference count reaches zero, the release() function deletes the instance. In order for a class to support reference counting through the POCO library, it must have the RefCountedObject as one of its base classes. The AutoPtr class acts as a container for a reference counted pointer to an instance. Instead of using primitive pointer types, we use AutoPtr instances to store pointers to a reference counted instance. The AutoPtr class automatically calls the duplicate() and release() functions of the instance when appropriate. Thus we use the AutoPtr instances and reference counting throughout the message passing system to ease memory management, to ensure that there is only one instance of a message in memory, and to ensure that the instance is destroyed when we no longer require it.

We have discussed using reference counting to memory manage message instances and share these instances among receivers. The second measure we take to reduce the number of message class instantiations during run-time is the creation of a generic object pooling system where we keep a number of instances of an object in memory. In the next section we discuss the use and implementation of the object pooling system in more detail.

4.2.1 Object Pool

We are discussing ways with which to reduce the number of message class instantiations when sending messages with the internal communication system of the engine. When numerous messages are sent in frequent intervals, we encountered performance issues, with excessive class instantiations and destructions as one of the causes. In this section

we introduce a generic pooling system where a number of instances of a class are kept in memory. The basic principle is that instead of creating a new instance during run-time we can fetch one from a pool, use some generic function to clear any existing data from the instance, use the instance and ultimately when we no longer require the instance, we return it to its pool to be re-used later. If the pool is implemented efficiently, we can save the computation time that would be used to allocate and free memory for instance creation and destruction.

Therefore, we must create a class that can store a list of instances of a specific class type. The class type that is stored must have a function that allows its data to be cleared. Then we can, instead of creating a new instance, use a pool to fetch an instance and clear it before we use it. The pool must also have a function that allows us to return an instance to the pool once we are finished with it. In order to achieve this in a generic way we define an abstract base class, namely PoolObject. This class provides a virtual function that should be implemented by deriving classes to clear an instance. Thus any classes that should be able to utilise an object pool must inherit from the PoolObject class. Every pool is represented by an instance of the ObjectPool class. The ObjectPool class provides the functions for getting an instance from the pool and returning an instance to the pool. Since the primary motivation for including such a pool in our engine is to use it to eliminate the time spent creating and destroying instances, it is important that the data structure and algorithm that our pool uses does indeed perform better than creating and destroying instances through the standard C++ new and delete operators. Preferably, the fetch and return operations in our pool should perform in constant time. That is, our fetch and return operations should be bound by $O(k)$ where k is a constant. There are a number of data structures and algorithms we can use to achieve this. One method would be to use a queue implemented with a linked list to store pre-allocated instances in the object pool. Both fetching and returning instances can be completed in constant time, if one keeps a pointer to the first and last elements in the linked list respectively. One issue, however, is that linked lists encapsulate their elements in structures that contain pointers to the next element in the list. These structures might need to be dynamically allocated, which is exactly what the pool should avoid. In our case we could avoid this issue by including the next pointer in the PoolObject base class. Another data structure that we can use is a stack implemented with an array. In this case both fetching and returning an instance to the pool also occurs in constant time. The only disadvantage with using an array is that it has a fixed size. However, we require that a maximum pool size is set for each ObjectPool. This is used to prevent the pool from keeping too many unused instances in memory. Therefore, the fixed size of the array is not a problem in this case. Hence, we use an array as a stack for the data structure in our ObjectPool class, since it does not have the container overhead of a linked list.

Some objects, in particular all message objects, in the engine must be able to be pooled but must also support reference counting. In order to achieve this, we created a modified RefCountedObject class that can have its reference counting behaviour modified by supplying it with an instance of ReferenceCounter. The ReferenceCounter class provides the abstract functions required for reference counting, namely duplicate(), which in the standard case increases the reference count by one, release() which decreases the reference count by one and generally destroys the reference counted instance when the reference count reaches zero. In the case of pooled objects we implemented PoolObjectReferenceCounter. The release() function in PoolObjectReferenceCounter decreases the reference count by one as in the standard case, but instead of destroying the object it returns it to its pool. The Message class is an example of a class in the engine that extends PoolObject and uses PoolObjectReferenceCounter for reference counting. We instantiate object pool classes for every type of message that we currently use in the engine.

Object pools have allowed us to reduce the computational overhead involved with memory allocation and instance creation when creating messages that are sent internally in the engine. However, in the case of the timer task, we found that it would still be a worthwhile optimisation to allow the kernel to send the time between frames directly to the tasks it executes, instead of using the message passing system to send a message about this time difference to every task.

In the next section we discuss some implementation issues around the input system and using the OIS library.

4.3 The Input System

We discussed the design of the general input system of the engine in Section 3.9. We use the OIS library to manage input from keyboards and mice. In this section we discuss some general implementation issues around the input system and in particular issues around using OIS.

The first implementation decision we had to make was whether to use buffered or unbuffered input with OIS. With buffered input one can configure OIS to call a function when input events from the keyboard or mouse occur. With unbuffered input, one has to periodically poll the state of each device to see whether input has occurred. The former method, buffered input, ties in well with the message passing communication model used in the engine. We could set up OIS to call functions in OISKeyboardInputDevice and OISMouseInputDevice instances when keyboard or mouse input occurs. From there InputMessage instances can be populated with the details of the input event and then sent to an input task. The latter method, unbuffered input, would require an input task

to periodically loop through a list of OIS input devices and poll each device for state changes. This method fits well with the general task structure that the engine utilises. Therefore the decision would have to be based on the processing involved when polling and checking device states and populating and sending messages. Ultimately we opted to use buffered input, utilising the message passing system, since it involves less processing by an input task and only generates messages when input events actually occur.

The second major implementation challenge was to determine how to implement the InputMessage class, what information to include and to ensure that the class is extendable by input plug-ins that could be added to the engine in the future. As discussed in Section 3.9, the input system of the engine generates strings to identify certain input events that occur. If the "W" key on the keyboard is pressed, for example, then the input system generates a string "keyboard0.W". The first part of the string identifies the device on which the event occurred and the second part identifies the event that occurred. These events strings are generated for any type of event. For example, if a mouse is moved horizontally, an event string such as "mouse0.X_AXIS" is generated. Since this is a fundamental manner in which the input system identifies an event that occurred on a device, this event string must be included in an InputMessage. An InputMessage must include all of the relevant information around an input event to maximise its usefulness. While the event string is a start, more information must be included. In the case of devices with buttons, an InputMessage must contain a field indicating whether a button was pressed or released. Therefore, we include an integer based device action field in the InputMessage. Depending on the value, the message would indicate whether a button was pressed or released or whether movement on an axis occurred. InputMessages also contain a device state vector. This vector is populated with information about the current state of a device when the input event occurred. In the case of movement on an axis, this vector would contain the absolute and relative positions of all axes of the device. The InputMessage class also contains a number of predefined static variables that provide indices into the state vector to easily find the absolute and relative positions of a specific axis. For example, the static variable INDEX_X provides the index into the state vector for finding the absolute position of the X axis of the input device. Lastly, InputMessages contain a field for the input action that is mapped to the input event identifier (see Section 3.9). An input action allows us to define a map between events that occur on an input device and actions that should occur in a game. If the "W" key is used to move the avatar of the player forward, for example, then the input event identifier "keyboard0.W" would be mapped to the input action string that would cause the player to move forward, "walkForward" say. The input action string "walkForward" could then be mapped to a function. Thus, if the "walkForward" action occurs, the engine calls this function. This target function can be defined in a C++ library or in a scripting language file.

With the InputMessage class implementation we tried to balance the quality of information included in the message, whilst striving to keep the class extendable and not too large, in terms of data size, so as to cause problems with high frequencies of input events.

In the next section we briefly the discuss the implementation of an audio plug-in for the engine.

4.4 Audio Plug-in

In Section 3.10 we discussed the design of the audio framework in the engine and also mentioned how our design is similar to that of the OgreOggSound plug-in. In this section we briefly discuss a plug-in for our engine, that uses our audio framework and OgreOggSound, to add audio support to our engine.

OgreOggSound is currently the only functioning audio plug-in for OGRE. It provides OGRE with positional audio by using the OpenAL [16] library. It is thus the obvious choice to use in our engine. The plug-in we implemented simply provides implementations of the StaticSound, StreamSound and SoundManager classes which all wrap the functionality of OgreOggSound. It was necessary to implement a plug-in for OgreOggSound for our engine (instead of just using it directly with OGRE) so that OgreOggSound could be properly initialised and managed through the structures of the engine.

In this section we briefly discussed implementing a plug-in for OgreOggSound for our engine. In the following section we discuss the implementation of a data capturing system that provides an SQLite implementation of the framework discussed in Section 3.12.

4.5 SQLite Data Capturing

Therapists and teachers indicated that a data capturing system would be useful since it will give them a way to keep track of the progress of users. In Section 3.12 we discussed the design of a framework in the engine that can be used to implement data capturing systems. The data capturing framework is reasonably generic, allowing one to create implementations in a number of ways. The requirements of an implementation for the data capturing system in the engine is that it must support methods for finding entities by their names, and finding entries in entities by using their key values. This is essential, since we must provide methods where developers can extract information from captured data in order to present such information to therapists through a graphical user interface. Read, write and search operations of an implementation must be relatively fast since the capturing and retrieving of data could potentially occur during an interactive game or

therapy tool. Thus we must consider existing methods for storing and representing data that are compatible with the design of our data capturing system. In Section 3.12 it was mentioned that our entity and entry model, with keys that uniquely identify entries, is similar in many respects to relational databases. This leads us to consider an SQL-enabled relational database server as a possibility for a storage medium for our data capturing system. An SQL based database implementation satisfies all of our requirements. Its organisational structure is similar to the entity and entry structure we use. Furthermore, an SQL based database would allow us to use SQL queries in our data capturing implementation in order to store and fetch data. The storage structure of relational databases are generally designed to allow for quick fetching of rows from a table by using keys.

Another possibility for a storage medium would be a storage system based on the extensible markup language [1] (XML). One could store information in XML text files or a native XML database. XML elements could be used to define an entity and its attributes. Individual entries could be stored in individual XML elements. However, our entity and entry model more closely resembles a relational database model. Therefore, we opted to use a well-known SQL-enabled relational database, namely SQLite. SQL queries and statements make it a simple task to implement all of the read, writing and searching functionality that we require from our data capturing system. We use SQLite instead of other SQL servers such as MySQL or PostgreSQL since it is a compact C library that is easy to include in an application, and it uses a local file for database storage. Thus we do not have to take network issues into account at this stage, although the generic data capturing framework does provide possible ways for handling remote databases.

One of the most important implementation considerations around the SQLite data capturing system involves whether to use delayed reading and writing to the underlying database, or whether to do immediate reads and writes. The primary disadvantage of using delayed operations is that the user must wait until some form of notification is received indicating that the operation has completed, before data read from the database is available, for example. The advantage of delayed operations is that control can be exercised by the engine on when these operations occur and how many of these operations occur at a time, or rather, how much time is spent on these operations. With immediate operations it is possible that a high frequency of operations, that are all executed immediately, can cause performance degradation and undermine the experience of a user playing a game or utilising the virtual world.

In our implementation of the SQLite data capturing system we use delayed operations. We feel the improved performance control that can be applied, along with the fact that it naturally favours a message passing communication system, makes it the logical choice. The SQLite based data capturing system consists of a number of classes that extend the classes discussed in Section 3.12, namely, SQLiteGameLogManager that is an extension of

the GameLogManager class, SQLiteGameLogEntity that is an implementation of the Entity abstract class and finally SQLiteGameLogTask which is a task. When the SQLite data capturing system is initialised, the SQLiteGameLogTask is inserted into the kernel. The SQLiteGameLogTask is responsible for conducting the read and write operations. Developers can use the SQLiteGameLogManager to read entities from, or commit entities to, an existing database. The SQLiteGameLogManager sends messages to the SQLiteGameLogTask and the task processes these messages when it is next updated by the kernel.

When users request read or write operations through the SQLiteGameLogManager, they receive a unique identifier for each operation. Users must be able to determine whether an operation has been completed. This can be accomplished by using the SQLiteGameLogManager to check whether an operation has completed by using its unique identifier.

In this section we discussed an implementation of our data capturing framework from Section 3.12. This implementation uses a relational database, namely SQLite, with delayed read and write operations.

In the next section we discuss some implementation issues and decisions around the networking system in the engine.

4.6 Network

The design of the basic network model used by the engine is discussed in Section 3.12. In that model we transfer InputMessages generated on one host to all the other connected peers, simulating the action that a user initiated on a host on all of its peers. Consider the example where the user presses a button that generates the "walkForward" input action. This action is now sent to the connected peers of the host. Thus the peers receive the InputMessage containing the action "walkForward." This can be problematic if the peers do not know to which host this message actually belongs. Consider a simple virtual world where each player is represented by an avatar. Each player runs the same version of the virtual world. Hence, for the virtual world of each player the "walkForward" action must cause the avatar of the local player to move forward. When a message is received over the network that simply contains the "walkForward" action it is not clear which avatar (the local avatar or the one of the connected peer) should be affected by it. Our initial solution to this problem was to create a field in the Message class that identifies the creator of that message. With this information about the creator, it would be possible for the engine to know whether a message is local or not and could then possibly identify to which objects an action should or should not be applied. Ultimately the engine would also need to know which objects in the virtual world were created due to requests over the network, in order to be able to apply the action to the correct objects. Another more transparent approach

is simply to apply a different naming scheme to objects and actions: before a message is sent across the network, all the name fields and action strings of any encapsulated messages or objects are prefixed with some unique identifier for the originating host. Thus the remote owner of an object or message can be identified by using the prefix of the name of the object. With respect to our "walkForward" example, when a host receives the InputMessage instance the action would be "<uniqueID>.walkForward" instead of just "walkForward". One of the reasons why this is a superior solution is because objects in the game world are generally controlled with Controller[2] instances that are called upon the reception of certain action strings (see Section 3.9). If we continue with our simple virtual world with avatars example, when a new player connects to the networked session, messages are sent to all peers informing them of the connection. Messages are also sent so that the avatar of the player and all relevant Controller instances and action mappings that are used to control the avatar are created at all peers. This information is sent by using the Serialisable interface to serialise the relevant Controller and GameCharacterObject instances. In this case we also apply the naming convention to all instance names and action strings. Thus the controller that is responsible for moving the avatar responds to the action string "<uniqueID>.walkForward" on remote hosts instead of just "walkForward" as it does on its local host. This solves the problem of determining to which objects remote actions belong. However, this solution is only transparent and reasonably automatic if a game makes use of Controller instances for game objects that are controllable by a user. If an implementation does not use Controllers and instead registers to receive InputMessages directly, then that implementation would have to inspect the messages, recognise whether the message is local or from a remote peer, and then take appropriate action. This provides developers with flexibility, but it could also prove to be problematic for developers that are unaware of the Controller paradigm.

Currently the unique identifier of a host is the IP address of the host. This might not be sufficient in all cases, such as situations involving network address translation or having multiple instances that are running on one computer. Instead of the IP address one could use an algorithm to generate a universally unique identifier to use as the unique identifier for the host.

In this section we discussed how we use instance and action renaming to successfully transfer input actions among networked peers.

In the following section we discuss the implementation of scripting language support in the engine.

[2]Controller classes are used to provide a mechanism through which the movement of GameActorObjects can easily be controlled. Generally these classes are convenience classes used to bind action strings to move or rotate GameActorObjects.

4.7 Scripting

One of the objectives of this project is to provide a platform for educational game creation and ASD therapy tool creation without the need for extensive programming knowledge. One of the first steps necessary to accomplish this objective is to make the lower-level C++ API accessible in high-level scripting languages. Our language of choice, for its relative simplicity and speed, is Lua [31].

There are a number of tools available to simplify the process of linking a C or C++ application with a scripting language. For Lua specifically, one could use the standard Lua C/C++ API, tolua++ [38] or luabind [48]. The simplified wrapper and interface generator (SWIG) [10] is another tool that allows one to utilise a library written in C/C++ from a scripting language. SWIG allows one to easily link a C or C++ application to many different scripting languages, not just Lua. The process involves the creation of interface files that define the API of the C or C++ application to be wrapped to the scripting language. These interface files are so-called "cleaned" versions of C++ header files. Functions that should not be available to the scripting language are removed when the C++ header file for a class is copied to a SWIG interface file. It is also possible for SWIG to directly use uncleaned C++ header files. Tolua++ also supports such interface files, but we found that it has far inferior C/C++ preprocessing capabilities and requires many modifications from the original C++ header files. Luabind does not support interface files, and one must use the Luabind C++ API to manually specify which classes are wrapped to the scripting language. The use of interface files highlights an important question: how much of the C++ API of the engine should be wrapped to a scripting language? The answer ultimately depends on what one wants to achieve. If the entire API is wrapped then one can use all of the functionality of the engine from a scripting language, which in itself is a desirable feature. The risk would be less control over what end-users might accomplish in terms of modifying games or tools and causing unwanted behaviour by accessing the scripting system while engaged in a game.

We chose to expose the majority of the engine API to the scripting languages, making it possible to write a full application, that utilises the engine as a library, from a scripting language. We use SWIG for the wrapping, since it has better support for directly parsing C++ header files and it supports the generation of wrappers for multiple scripting languages. We include the Lua wrapper library directly in the engine library since we selected Lua as our primary scripting language. Other scripting language libraries are implemented as plug-ins. We have implemented a plug-in for scripting support in Java. In the following section we discuss the implementation of plug-ins for scripting language support in the engine.

4.7.1 Plug-ins for Scripting Languages

In this section we discuss some of the implementation requirements for plug-ins that add support for scripting languages to the engine. In particular we implemented such a plug-in for Java. When the initialise() function of the plug-in is called, it must load the wrapper library for the engine into the environment of the scripting language. SWIG provides a function for this in the wrapper library it generates, called SWIG_init().

Scripting language plug-ins must provide a functor (a functor is a class that wraps a function call; see Section A.3) implementation for function calls to the scripting language from the engine library. SWIG provides the wrapper library allowing the scripting language to call functions in the engine library, but we also require the engine to be able to call functions defined in the scripting language. The message passing system uses the functors defined by scripting language plug-ins to deliver messages to functions in the scripting environment. We also require it for event callbacks, most notably input action event callbacks to scripted functions. A functor allows us to hide the implementation detail behind calling a function defined in a scripting language from C++, since this would entail interacting with the C++ API of the scripting language in question. Thus we need to implement a framework in the engine for functors. This framework needs to be extendable by scripting plug-ins in order to support functors for specific scripting languages. But we also require an existing class from the functor framework to use throughout the engine where message delivery or event callbacks to script functions can occur. We therefore need to build a functor hierarchy with a generic and extendable base class. The base class is used throughout the engine and would define some virtual functions. Plug-ins can then extend the base class and implement the virtual functions and add additional functionality. The framework consists of an abstract base class called FunctorBase. The FunctorBase class defines virtual functions for setting parameters to the wrapped function as well as calling the wrapped function. All systems in the engine that support functors, for instance message delivery and event callbacks, use pointers or references to FunctorBase instances. This allows scripting language plug-ins to extend the abstract FunctorBase class by creating new functor classes through inheritance. These new functors would then contain the required functionality to call functions from the scripting language in question. Thus we have a hierarchy of functor classes with a generic base class, FunctorBase, where the hierarchy can be extended by scripting language plug-ins. Given how plug-ins can provide functors to allow the engine to call scripting language functions, we must discuss how to manage the instantiation and deletion of functors. We need to investigate whether it is necessary to provide a framework for the instantiation of functor instances, or whether we can simply allow developers to create the appropriate instances manually in their scripts as necessary. Firstly the FunctorBase class acts as the base class for all members of the functor hierarchy. All engine functions that can support

functors only require references to FunctorBase instances. This level of abstraction is similar to the GameObject hierarchy in Section 3.6. Hence the abstract factory design pattern is also applicable in this case. Furthermore, there are potential memory management issues when creating functor instances in a scripting language with garbage collection and passing those instances to the engine; this is discussed in detail in the next section. Thus using a factory provides us with a central place to create functor instances as well as a way to solve the memory management issues discussed in the next section. Therefore, we include the FunctorFactory class as an abstract factory in our functor framework.

In the next section we discuss a memory management issue we encountered in using the engine as a library from scripting languages.

4.7.2 Memory Management

Many scripting language environments have built-in garbage collectors and reference counting for memory management. We found that in certain situations this can lead to memory management problems when using the scripting languages with the engine. A problem that we encountered is the following: one creates an instance of an object in the scripting language, pass it to the engine through a function call, the engine takes ownership of the instance, and frees it when necessary. The scripting language environment, however, would not be aware of the fact that the engine library has taken ownership of the instance and would at some point in the future, through garbage collection, free the instance if there were no more references to the instance in the scripting language environment. This does not necessarily need to be a case where the engine takes ownership of the instance. If one passes an instance to the engine without keeping a reference in the scripting language environment, garbage collection would destroy the instance, while the engine is still possibly referencing it. An example of where this occurred was with the use of functors. One would create a functor, in the scripting language, to use with an input action callback. The functor is then passed to the engine, but all references to it in the scripting language eventually disappear since we no longer need to access it from the scripting language once it has been passed to the engine. The garbage collector of the scripting language would then free the functor instance, which is shared with the engine library, at some point in the future. As soon as the appropriate input action is then fired, an error would occur when the engine tries to reference the destroyed functor. One solution would be to simply always keep references to such instances in the scripting language environment, but this is more of a work-around than a solution. Another solution would be to, in the event that the engine takes ownership of the instance, use a disown typemap in SWIG. The disown typemap allows us to specify which C++ functions take ownership of its parameters. When SWIG generates the wrapper library, it then

ensures that the garbage collector of the scripting language will not destroy an instance that has been passed to the engine via a function that has been marked with the disown typemap. There are two problems with this approach. The first problem is that it requires us to locate every single function where this might occur, manually, and apply the disown typemap to it. The second problem is that SWIG does not currently support the disown typemap for all of its target scripting languages. In particular, it is not supported for Java. This problem can also be solved by ensuring that objects are created by using the engine and can then be properly memory managed by the engine. In the specific case of functors we would require the scripting language plug-in to define a factory to create its functor instances. One would then use that factory instead of manually creating the instance. This ensures that the scripting language environment never has ownership of the instance and as such garbage collection issues cannot occur. It is also a better way to memory manage instances in general and uses the factory design pattern that is common to this engine.

In this section, we discussed the implementation of scripting language plug-ins for the engine. We discussed functors that allow the engine to call functions defined in the scripting language. Finally, we mentioned measures taken to resolve memory management problems between garbage collection in scripting languages and the engine.

In the next section we discuss the implementation of the system used to provide support for displaying text in multiple languages.

4.8 Localisation

South Africa is a country with multiple cultures and multiple official languages. It is therefore important that educational and therapy tools be available in as many languages as possible. In general the process of localising a game involves more than just language. Graphical and story elements can be culturally specific and a localisation process should also consider such aspects [56]. However, at this stage we only consider basic language localisation. Thus, our engine should support translation of game implementations to different languages. In this section, we discuss the implementation of such a language localisation system for the engine. We begin by discussing the GNU *gettext* [29] library that is used to provide support for multiple languages in many open-source applications. We discuss certain difficulties we would have in using it in conjunction with the scripting interface, and finally we then discuss our solution.

GNU *gettext* is an open-source library that is used to provide support for multiple languages in applications. One sets up translation files that define translations for certain keys (segments of untranslated text). For each translation one can also include a context

description, since context differences can result in major differences in ultimate meaning. The translation file can also include information about how to translate a key in singular or plural form. Many editors, and plug-ins for existing text editors, are also available that can be used to easily edit existing translation files or create new translation files. GNU *gettext* is a C library that provides functions and C preprocessor macros that is used in the C source code of an application. All strings that should be translated must be replaced with a *gettext* macro that handles the translation.

```
1 msgid Name
2 msgstr Naam
3
4 msgid Close
5 msgstr Maak toe
6
7 msgid Ok
8 msgstr Reg
```

Listing 4.1: A portion of a translation file for English to Afrikaans.

In our case, many games or tools built on our engine will be almost completely programmed in a scripting language. This complicates the use of *gettext*, since it would have to be included in the scripting language wrapper library. Furthermore, *gettext* makes use of preprocessor macros that we would not be able to wrap directly with SWIG. We therefore decided to implement a simple translation system that maps an untranslated string and its context to its translated version. This is a fairly simple implementation that relies on the map class provided by the C++ standard template library (STL). We kept the file format as close as possible to the format of *gettext* translation files. This ensures that switching to *gettext* would be possible in the future, if we can find a better way to integrate it into the scripting language wrapper interface. This solution provides us with the functionality that we require and is reasonably simple.

The localisation system consists of a Translation class that manages the map of untranslated to translated strings for a single language. A singleton, DefaultTranslation, exists that is generally used as the current active translation for the engine. We included a number of functions in a helper library for Lua, with the goal of simplifying the use of the translation system from Lua.

In this section, we discussed the language localisation component of the engine. This allows one to provide games and therapy tools in multiple languages.

In the next section we discuss how 3D models and scenes are created for our engine and how these models and scenes are loaded into our engine.

4.9 Content Creation and Loading

So far we have discussed most design and implementation aspects surrounding the engine. What we have not discussed, however, is how to go about creating scenes and 3D models that the engine can render. Since we use OGRE, we must use tools and file formats that are supported by OGRE. OGRE uses its own file format for 3D models, or meshes, and for the skeletons of characters. The OGRE community maintains a mesh exporter for the open-source 3D modelling and animation package Blender. This exporter can export any 3D object created in Blender to a file that can be loaded by OGRE. This includes avatars with skeletal animations. The mesh exporter is limited to exporting single objects, one at a time. To export a scene, the OGRE community also maintains a scene layout exporter for Blender and importer for OGRE. This scene layout format is called Dotscene. The Dotscene file format is XML based. Thus, we can create a scene in Blender, for instance the interior of a restaurant, and export the objects in the scene as well as the scene layout using the OGRE mesh exporter and Dotscene exporter for Blender. Using Blender to create 3D models is a time consuming and difficult task. Therefore, we acquired a number of free 3D models on-line from the following repository: `http://www.katorlegaz.com/3d_models/index.php`. These models were released under the *Creative Commons Attribution 3.0* license (see `http://creativecommons.org/licenses/by/3.0/us/`).

In order to correctly load Dotscene files exported from Blender into our engine, we had to extend the existing Dotscene importer for OGRE. We had to extend the loader so that it does not just create OGRE entities, but instead use our GameObjectFactory implementation to create GameBodyObjects and GameCharacterObjects. This proved to be a simple task.

In this chapter we discussed various implementation issues and aspects regarding the design discussed in Chapter 3. In the next chapter we discuss the implementation of an educational game and two therapy tools using our engine.

Chapter 5

Educational Game and Therapy Tool Implementations

In this chapter we discuss educational games and therapy tools we built using the engine. These implementations were built specifically to test the features and performance of the engine. We discuss three implementations. The first is a computerised version of portions of a workbook for individuals with ASD. The second implementation is a basic educational game that was developed and tested in conjunction with a local school. The third and final implementation is that of a social skills therapy tool similar to those discussed in [43, 44]. During our discussion of these implementations we also discuss some issues encountered with the engine while developing these implementations and how we resolved said issues.

In the following section we discuss the implementation of a computerised version of *I Am Special* [66].

5.1 I Am Special

In this section we discuss the implementation of a computerised version of *I Am Special*, a workbook used during therapy for children with ASD. The book contains a number of worksheets that participants fill in over a period of time. In this implementation we computerised a small selection of these worksheets.

The implementation relied on the GUI component of the engine and the associated GUI layout editor. For each computerised worksheet, artwork and a GUI layout were created. An example of a computerised worksheet is shown in Figure 5.1.

Upon launching the program, users are presented with a login screen where they have to log in using their names. After logging in, they can navigate to any worksheet and

Figure 5.1: A computerised worksheet from *I Am Special*.

fill in values or change values they have previously filled in. Users are allowed to fill in anything, and the system does not perform any context checking on values that are entered. Therapists and teachers indicated that it would be more valuable for them to see what the children entered, and then be able to discuss it with them, than having the system forbid a child from entering something.

All the worksheets have speech guidance. If a user clicks on a label or text box, an audio description of the required input is given. The language of the worksheets, including text and audio, can be set to Afrikaans or English, depending on the preference of the user. The worksheets use the language localisation system discussed in Section 4.8 for text, making it possible to easily add more translations in the future.

This implementation stores all of the values that the users enter into computerised worksheets using the SQLite data capturing system of the engine. All entered values are attributed to a specific user by using the name of the user. If a user returns at a later stage and logs in using their name, the data they previously entered is loaded and displayed on the computerised worksheets as they navigate through them.

Development of the computerised worksheets proceeded mainly smoothly. We encountered a minor problem with reading certain special characters from translation files, but this was resolved by switching the encoding of translation files to UTF-8.

This implementation was created and tested in a week and was programmed in Lua. Thus this implementation uses features from, and tests the following aspects of, the engine:

- the scripting language interface, specifically Lua;
- audio;
- the text translation system;
- the data capturing system;
- the GUI system and its layout editor;
- the input system of the engine; and
- the task management and communication interface.

In the next section we discuss the implementation of a basic educational game implemented with the engine.

5.2 Journey to the Moon

We implemented a basic educational game, aimed at grade one learners, in partnership with a local school. In this section we discuss the implementation, structure and content of the game, how the state of the game is related to a finite state machine (FSM) and we discuss a plug-in we created for the engine to manage a system of categorised questions and answers that can be used in educational games.

We met with a teacher from the school to discuss the general educational content of the game [11]. It was decided to present learners with the same types of problems in the game that they would encounter in the normal class situation. Players of the game do not control a character directly. Instead a character in the game communicates the events of the game to the player through text and audio. Since the game is aimed at grade one learners, it was important that all text in the game also had associated speech, since not all of the learners would be able to read at a satisfactory level yet. Furthermore, all text and audio in the game are available in both Afrikaans and English. Text in multiple languages is provided by using the language localisation component in the engine.

In general the character in the game presents the player with some story elements and then requests the player to help him solve some problems. These problems are typically arithmetic or linguistic exercises, similar to what the learners would encounter in class. When the learners solve these problems successfully, they are rewarded with an animation sequence in the game. The game is relatively short, consisting of only two scenes. The game is designed to provide players with structure and clear goals. Such scaffolding is important in educational VEs [34].

We now describe the game in more detail. The game is about a scientist who receives a message requesting him to go and investigate something mysterious on the moon. The first scene is set in his laboratory. There he discovers that a malfunction is plaguing all of his equipment: he has to solve certain arithmetic and linguistic exercises in order to operate his equipment. This is where the player enters the story. The scientist enlists the help of the player in solving these problems. In the first scene the scientist must prepare his ship for travel to the moon. This requires opening the hatch to the bay where his ship is docked, opening the exterior door of his laboratory and finally launching the ship. As the player complete exercises, each of these events occur with an animation. Figures 5.2 through 5.4 show the scene before the ship is released from its dock, before the exterior door is opened and finally show the ship during take-off.

In order for each event to occur, the player must complete five exercises correctly. The first criticism of this game is the fact that during this initial scene there is no limit on incorrect answers: the game will continue presenting the player with new exercises until five correct answers have been given. The same exercise is repeated three times if answered incorrectly, before moving on to a new exercise. This can become frustrating to players that are struggling to answer correctly.

Once the ship has launched, the second scene starts. In this scene the scientist explains that they are encountering asteroids on their way to the moon. The player must help the scientist to make quick calculations in order to pilot the ship around an asteroid. The ship approaches an asteroid and the player has a limited amount of time in which to solve a problem in order to dodge an asteroid. If the player takes too long to solve the problem, the ship hits an asteroid. The player is informed that the shields of the ship can only take three hits before they would have to turn around and restart the scene. Players have to solve one extra problem per asteroid, so this scene becomes progressively more difficult. Three asteroids are dodged before a cluster of asteroids is encountered. In this case the scientist destroys an asteroid instead of dodging it. After the ship clears the asteroid field, the scientist informs the player that she must now help him to prepare the landing sequence for landing on the moon. This involves another set of problems that must be solved before a landing animation is played.

From this discussion of the scenes in the game we can see that certain game states emerge.

Figure 5.2: Before the ship is released from its dock.

Figure 5.3: Before the exterior door is opened.

Figure 5.4: During ship take-off.

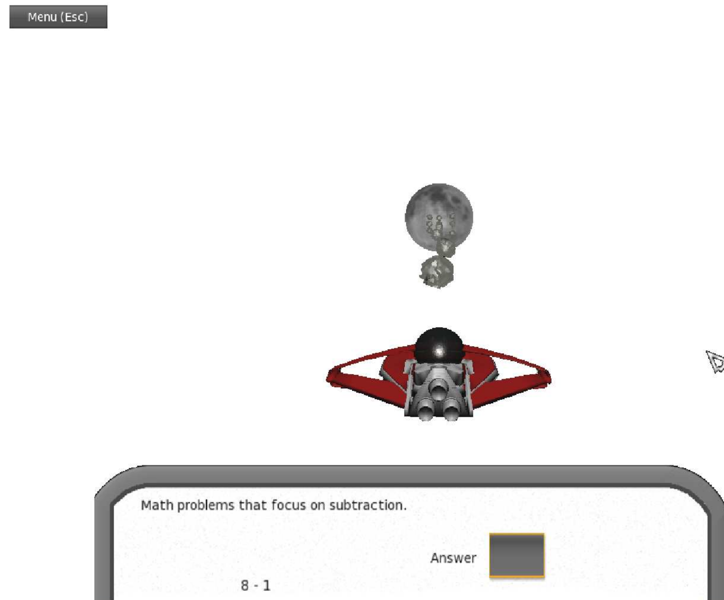


Figure 5.5: The start of the second scene.

Figure 5.6: Dodging an asteroid.

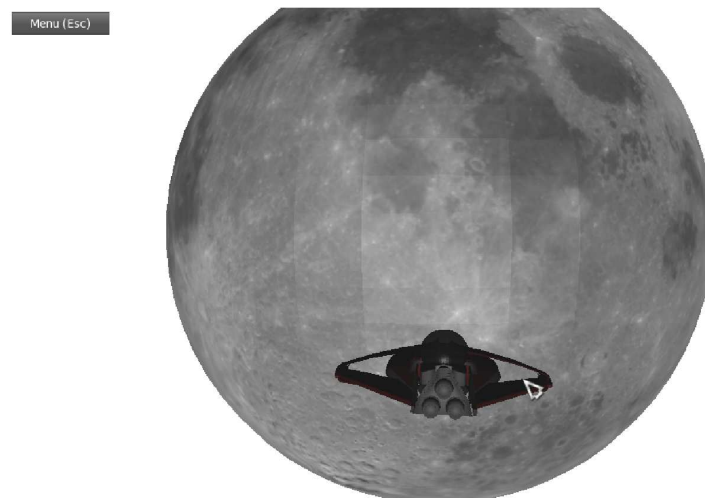


Figure 5.7: Approaching the Moon.

For instance, a state in which the scientist is talking with the player, and then a state in which the player is charged with solving problems and a state in which animations are played. It is useful to think of sets of variables in the game that are represented by states. Events that cause these variables to change are the transformations from one state to another. Using this model we can give a high level representation of the game logic as a finite state machine (FSM). In Figure 5.8 we give a portion of a state machine that could potentially represent the game logic of this game. Each numbered state corresponds to an unique state assignment of the set of game logic variables. Examples of game logic variables could be a variable that indicates whether the scientist is speaking or not, variables indicating which speeches have been given, which problems have been solved and which animations have been played.

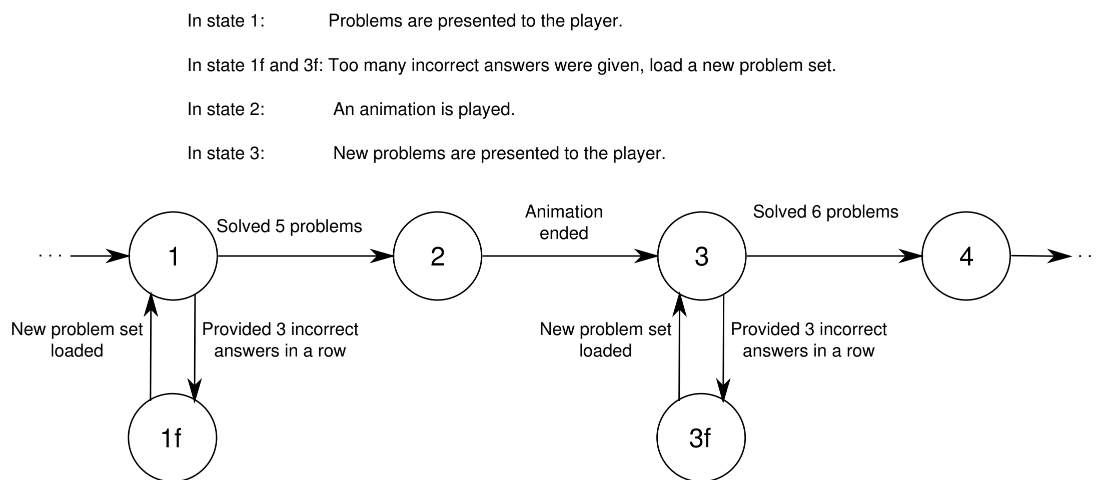


Figure 5.8: A portion of a FSM that could represent the game logic of the *Journey to the Moon* game.

This game does not make use of the replay or networking features of the engine, but it does use the data capturing system. Players identify themselves by using their names. The system captures all the problems that they are presented with, and the answers that they give to these problems. A GUI exists that teachers can access to view all of the sessions that any player has played. There the teachers can see all the problems that they were presented with and their answers to those problems. The teachers that we showed this to were very positive about its use. It allows them to identify with what types of problems, or with what specific problems, a child struggles. For this game the data capturing system stores the names of all players in a table named “Players”. Names are considered to be unique. This should be improved in future, since it is possible for more than one individual to have the same name, but it is not a massive problem in the relatively small classes where the software was tested. Each time a player logs in, a new session identifier is created in the “Sessions” table. This session identifier is merely the name of the player and the current time stamp. Finally there is an “Answers” table in

which the details about a presented problem, and the answer that the player provided, is stored along with the session identifier. Over time this table can grow to be quite large. A future improvement would be to consider archiving content on monthly intervals, although in a practical situation new databases would be created every year for a new class, so this might not be too problematic.

The game logic was implemented using Lua. It became obvious that it would be advantageous to have a game logic function that is called by the engine at regular intervals. Although we can be alerted to changes in game state through messages and callbacks, there are certain situations where we might need to continually check and update an object in the game. An example would be in the second scene where the ship of the scientist moves toward the moon. We could have implemented a Controller class in the engine to control the ship, but since the movement is relatively simple, and the manner in which the ship moves is reliant on the current game state, we can also use a regularly called function in the scripting language to continuously move the ship forward while certain game state conditions hold. One manner in which to achieve this is to use a task. The engine kernel updates all tasks regularly. Since our game logic implementation is in Lua the task should also be created in Lua. This presents us with a problem when connecting our engine to scripting languages and using abstract base classes with polymorphism. We have to create a new task type in the scripting language, that inherits from the Task class in the engine (through the script wrapper interface) and we provide implementations for the abstract functions. The problem with this is that there is no way for the engine library to know that an implementation of the class has been provided in the scripting language. Therefore, when the kernel calls a function from the Task instance the version in the scripting language will not be called, since polymorphism does not extend between the library implemented in C++ and the scripting language interface. The intention of the scripting language was not to allow for in-depth extensions of the engine library, but rather a manner in which implement and test game logic quicker and to keep game logic separate from the engine library. In order to overcome this issue we implemented a new task in the engine library, namely the CallbackTask. In this task functors can be used to define functions that should be called for each of the virtual functions in the Task class. Thus we can set up a CallbackTask instance from Lua and configure it to call specific Lua functions when the task is initialised, suspended, killed and executed.

Both scenes in the game were created in Blender. Creation of the first scene and implementing its logic in Lua occurred over the course of about one week. A considerable portion of this time was spent in assembling the scene in Blender and finding suitable 3D models from on-line repositories. In contrast, creating the second scene took about two days. Therefore, the creation of 3D content and scenes would appear to be a problem at this stage. Using Blender to create the scenes can be somewhat difficult and time

consuming. It would be advantageous if an easier to use tool could be implemented in the future that would allow for easy creation of scenes by using 3D models from existing repositories.

The game was tested at a local school for 20 days. During this time the game and engine did not suffer any errors or crashes. There was one technical issue where the operating system of the computer on which the game was installed failed to boot, but this issue was unrelated to our game and engine. We supplied the teacher of the class, in which the game was tested, with a questionnaire posing several questions about the usefulness and relevance of the game. The survey is attached in Appendix B. The response from the survey was positive. The learners appeared to enjoy the game and was excited by the prospect of using basic linguistic and arithmetic skills to progress the story of the game. The teacher was of the opinion that the game helps the learners improve their arithmetic and linguistic skills. Criticisms were mainly aimed at GUI elements that were too small or lacked pictures and contained too much text.

The game tests the following aspects of the engine:

- 3D content and scene loading,
- rendering,
- the game object hierarchy,
- the physics and collision detection system,
- audio,
- language localisation,
- input handling,
- scripting language interface (Lua specifically),
- SQLite based data capturing,
- the GUI system, and
- the task management and communication interface.

This game in essence relies on a system that asks a player a question and the player must provide the correct answer. In order to ease the creation of future games and the extension of existing games we implemented a system that loads question category files from files in an XML-like format. We discuss the design and implementation of that system in the following section.

5.2.1 The Problem Category Library

We have to implement a system whereby sets of problems can be defined in files. These files should be text files so that it is easy to add new problems and categories in the future. In order to provide a human readable and logical structure to the content of the files and to also make the files easy to parse and load by the engine, we use an XML-like file format. The POCO library has support for parsing XML files. In order to further test the engine we implement the problem category library as a plug-in for the engine.

In listing 5.1 we show an example of a problem category file. In this case the problems are words from the English language that must be completed by filling in the missing letter. Each file can contain only one category, enclosed in the `<category>` tag. Inside the `<category>` tag there can be tags for a name and a description of the category, `<name>` and `<description>` respectively. Inside the `<category>` tag many `<problem>` tags can be defined, each representing a different problem or question. The `<problem>` tag can also have tags for a name and a description, as well as a `<question>` tag that contains a string that states the question as it should be presented to someone that must provide an answer. The `<correctAnswer>` tag is used to indicate what is the correct answer to the particular question. It is possible for a problem to have many `<correctAnswer>` tags. The manner in which a supplied answer would be considered correct is defined by the `<answerCondition>` tag: this could be “oneOf” which indicates that a player needs to give at least one of the answers defined as a `<correctAnswer>`, if it is “allNoOrder” the player must supply all of the answers defined with `<correctAnswer>` tags and if it is “exact” the player must give all of the answers in the same order as in the file. Problems could also have a number of `<possibleAnswer>` tags. Each `<possibleAnswer>` tag defines a possible answer to the problem that would be used in multiple choice questions.

```
1  <category>
2    <name>Language: Fill in the missing letter</name>
3    <description>Fill in the missing letter.</description>
4    <problem>
5      <name>Question 1</name>
6      <description/>
7      <question>p_t</question>
8      <correctAnswer>o</correctAnswer>
9    </problem>
10   <problem>
11     <name>Question 2</name>
12     <description/>
13     <question>p_n</question>
14     <correctAnswer>a</correctAnswer>
```

```
15        <correctAnswer>e</correctAnswer>
16        <answerCondition>oneOf</answerCondition>
17     </problem>
18  </category>
```

Listing 5.1: A portion of a problem category file.

In order to load such a problem category file, we use the XMLConfiguration class in the POCO library. The class loads an XML-like file into a standard string-to-string map. It accomplishes this by concatenating tags within tags to form keys. This makes it relatively simple to parse and load problem sets from these XML-like files. An example, from listing 5.1, would be the key "category.name" for the `<name>` tag inside the `<category>` tag. The string value that the "category.name" key would map to in this case would be "Language: Fill in the missing letter". In cases where there where a tag is repeated, such as with the `<problem>` tag, keys are created with a numeric index in brackets concatenated to the end, for example "category.problem[1]". Thus we can extract the different problems and their properties by first testing whether a value exists for "category.problem.name" for the first problem, from there we continue with "category.problem[x].name" where $x \geq 1$, until "category.problem[x].name" does not exist in the map as a key. A similar process can be followed to test the existence of multiple `<correctAnswer>` and `<potentialAnswer>` tags.

We implemented a plug-in for the engine that manages problem categories. The plug-in consists of three primary classes, EducationalQuestionsManager, EducationalQuestionCategory and EducationalQuestion. The EducationalQuestionsManager class has a singleton instance that is responsible for managing all existing EducationalQuestionCategory instances. The EducationalQuestionCategory class has a function for loading a category from a file, using POCO. For each question in a category an EducationalQuestion instance is created, and this instance is managed by the appropriate EducationalQuestionCategory instance. The EducationalQuestion class has functions for retrieving relevant information, such as the question text, possible answers, correct answers and the answer conditions. It also has a virtual function, that has been implemented with basic functionality, for testing whether a given answer is correct by comparing it to the set of correct answers and taking into account the answer condition of the question. The EducationalQuestionsManager can be used to retrieve a list of categories and their associated EducationalQuestionCategory instances. EducationalQuestionCategory instances can then be used to obtain a list of EducationalQuestion instances. Thus tool and game implementations can use this plug-in to manage a fairly robust system of categorised questions.

In the following section we discuss the creation of a therapy tool aimed at teaching social skills to individuals with ASD.

5.3 Social Skills

In order to test the manner in which our engine would be useful in terms of creating ASD therapy tools, we implemented a basic version of such a tool. Parsons *et al.* [43, 44] investigated the use of computerised therapy tools for social skills training. In this section we discuss the implementation of a social skills therapy tool, using our engine, that is similar to that of Parsons *et al.*.

We opted to implement one social skills training scenario: a restaurant where the user has to order food, pay for it and then find a place to sit. This is similar to one of the scenarios used in [43]. Our restaurant scenario uses the replay system of the engine to record a replay of a session, as well as the data capturing system to store the choices that the user made. A GUI is available for use by therapists allowing them access to the data that was captured during a session and allowing them to view replays. Another feature that the restaurant scenario supports is remote monitoring using the networking components in the engine. Therapists can connect to a session in which a user is engaged, from another computer. At this stage therapists can only monitor a session and can not yet pause or interact with the session in any way.

A restaurant scene was created in Blender. The scene contains a number of tables and chairs, a counter from where food must be ordered and an employee behind the counter. A screen capture of the restaurant scene can be seen in Figure 5.9.

Figure 5.9: The restaurant scenario in the social skills application.

A player controls the camera by using the mouse and keyboard. At this point in time there

is no GUI available for changing the configuration of the controls, but the configuration is defined in text files that can be modified. The player receives instructions from text prompts at the top of the screen and also through voice prompts. The language of the scene, including all text and voice playback, can be set to either Afrikaans or English. The text is translated using the language localisation system of the engine, so that more languages can be added relatively easily. The overall goal for the player in this scenario is to go to the counter, order food, pay and find a place to sit. Currently, the restaurant scenario has only one difficulty level. At this level there are no other characters in the scene, except for the player and an employee behind the counter. Thus ordering food, paying and finding a seat is simple in the sense that there essentially are few incorrect actions that could be taken. In future more difficulty levels can be implemented. For example, Parsons *et al.* introduce a queue of patrons at the counter, at higher difficulty levels. The player must join the queue at the back and not jump the queue. Tables might be filled with computer controlled characters, so that the player must ask the characters at a table whether or not an open seat is available. Also one could add ambient sounds and characters that move around, slowly increasing the realism of the scenario.

While an individual is using the tool, a replay is recorded. Thus a therapist can access a GUI at a later stage to view these replays. Furthermore, a therapist can also connect to a session that is in progress and view the progress of the user. However, since the networking components do not yet support synchronisation the therapist must connect to the session at the beginning, before the user can take actions that could modify the state of the session. In our tests the replay and networking systems functioned well.

This basic tool tests the following features of the engine:

- 3D content and scene loading,
- rendering,
- the game object hierarchy,
- the physics and collision detection system,
- audio,
- language localisation,
- input handling,
- scripting language interface (Lua specifically),
- SQLite based data capturing,
- replays,

- networking,
- the GUI system, and
- the task management and communication interface.

In this chapter we discussed three programs specifically implemented to test the engine. These programs cover all of the engine features defined in our goals in Chapter 1. The test programs performed well and the educational game from Section 5.2 was successfully tested at a local school for 20 days with no failures.

In the next chapter we discuss the general test methods used during the development of the engine.

Chapter 6

Testing

In the previous chapter we discussed three implementations specifically developed to test the engine. In this chapter we discuss some of the general measures used to test the engine and how effective these measures are. We begin with a brief general overview on some of the ways in which to test large programs.

6.1 Overview

There are two common types of tests that could be used, namely, structural tests or functional tests [33, 46]. Structural testing focusses on the internal structure of a program to detect programming errors. Structural testing is also referred to as glass box or white box testing, since the tester generally has knowledge about the source code. Functional testing is used to test whether a program satisfies the functional requirements that were defined during its design. This type of testing is also referred to as black box testing, since the tester does not have knowledge about the source code. Generally one should include both types of tests when testing a program to ensure an acceptable level of testing.

Tests can further be classified as static or dynamic. Static analysis is performed without executing the program that is being tested, whereas dynamic analysis involves running tests during program execution and then collecting and examining these results when the program terminates. Examples of static analysis include checking for variables that are used without being initialised, or syntax checking.

Lastly tests can also be manual or automatic. Manual tests require human interaction where individuals run a number of tests separately. Automatic testing involves writing programs, or generating test programs from some requirements specification, to test a program.

During the initial implementation phase of our engine we conducted many structural tests and functional tests that focussed on the particular segment we were implementing at any point in time. However, due to time constraints, we could not maintain this level of testing. We therefore began to limit the number of structural tests and focus more on functional testing. We implemented a number of small applications that focussed on testing specific functionality in our engine, as defined in our goals in Chapter 1. In the following section we discuss these applications and the features that they test.

6.2 Functional Tests

	P1	P2	P3	P4	P5	P6	P7	P8	P9
Rendering	●	●	●	●	●	●	●	●	●
Animation		●	●		●			●	●
Tasks	●	●	●	●	●	●	●	●	●
Communication	●	●	●	●	●	●	●	●	●
GameObject Management	●	●	●	●	●			●	●
Input	●	●	●	●	●	●	●	●	●
Audio				●			●	●	●
Physics and Collision Detection	●	●	●	●	●			●	●
GUI	●	●	●	●	●	●	●	●	●
Lua Scripting				●			●	●	●
Data Capture						●	●	●	●
Replay									●
Network					●				●
Localisation							●	●	●

Table 6.1: Nine testing programs, P1–P9, with the engine features that are tested by these programs.

In this section we focus on nine particular programs that we implemented to test the functionality of the engine. We call these programs P1–P9. These programs are used as manual dynamic functional tests for the engine. The first program, P1, allows the user to create stacks of cubes, manipulate the camera with the mouse and shoot cubes. The second program, P2, tests controlling the movement of avatars and playing skeletal animations of avatars. The third test program, P3, tests the loading of scenes, exported from Blender, into the engine. P4 is a simple two-player spaceship game implemented through the Lua scripting interface. P5 is similar to P2, but allows two hosts to connect to one another over a network. The sixth test program, P6, tests the data capturing system of the engine through a simple GUI with fields that can be saved to and recalled from a database. The final three test programs were discussed in Chapter 5. The *I Am Special* program (see Section 5.1) is represented by P7. We represent the Journey to the Moon game (see Section 5.2) with P8. The social skills therapy tool (see Section 5.3) is

represented by P9. Table 6.1 shows which aspects of the engine is tested by each of the testing programs.

We ran these tests numerous times and visually confirmed whether a certain feature functions as intended. These test programs assisted us in finding a number of errors and issues in the engine implementation. The tests also enabled us to confirm whether systems in the engine, such as replays and networking, function as expected.

In the next sections we discuss some specific issues and conclusions drawn from testing the engine.

6.2.1 Lua Scripting

Programs P4 and P7–P9 were implemented as Lua scripts. In all cases the scripting interface worked well. The only major issue that was identified, was that of the garbage collection system in Lua possibly deleting an instance of a class that the engine is using. We addressed this issue in Section 4.7.2.

6.2.2 Replay

We tested the replay system with the social skills therapy tool. We found that when the engine is running at an acceptable frame rate (25 frames per second or higher) replay recording and playback performs adequately with no visible loss of synchronisation. However, we artificially reduced the performance of the engine by running numerous background tasks and found that replays do not always play back correctly when the engine is running with low frame rates. This is in line with our expectations from the design of the replay system in Section 3.14. Incorrect playback occurs if the frame rate of the engine is too low, since messages recorded in the replay cannot always be sent at the correct time intervals. Therefore, in the general case, playback of replays is mostly acceptable. To ensure, however, that replays always play back correctly, the synchronisation method proposed in Section 3.14 could be implemented in the future.

The replay system also works well with respect to GUIs. In the social skills tool the replay system records any interaction a user has with a GUI. At one point the user is prompted, through a GUI pop-up, to select food to eat for lunch. This selection process, including the exact mouse movement, is recorded and played back by the replay system.

In the following section we discuss the testing of the networking system in the engine.

6.2.3 Networking

The networking system was tested in program P5 and program P9, the social skills tool. The tests were performed on a reliable 100Mbps local area network. We found that we could run a networked session for a number of minutes before visible synchronisation issues occurred. We generally would move an avatar around and position it at a landmark in the virtual scene. We would then visually confirm the position of the avatar on both peers and also, using the in-game scripting system, find the coordinates of the avatars in the virtual world. From our tests we concluded that it would certainly be necessary to include some form of synchronisation to the networking model discussed in Section 3.15, as even on a reliable low latency network synchronisation issues did occur over a period of time. This is particularly noticeable when monitoring user interaction with the GUI over the network. We implemented basic synchronisation in the social skills tool by sending the updated position and orientation of the avatar, every five seconds, to the peer connected for monitoring purposes. This eliminated any visible synchronisation issues with respect to the avatar. Therefore, our tests indicate that our network system works as expected.

Our tests were limited to the UDP version of our network model since, due to time constraints, the TCP version is not yet fully functional.

In the next section we discuss the testing of the language localisation system in the engine.

6.2.4 Localisation

The language localisation feature of the engine was tested in P7–P9. We provided Afrikaans and English translations for all text in all three programs. In some cases, localised strings would not display any special characters. This was determined to be a GUI configuration related issue that we subsequently corrected.

In this section we discussed the functional tests that were used to test the engine. We specifically discussed the test results of a number of important systems in the engine. In most cases systems worked as expected or errors were identified and fixed, through the tests that were conducted.

In the next section we discuss how many functions in the engine were covered during testing.

	Function Coverage
Tasks	69%
Communication	77%
Data Capture	60%
Replay	61%
Network	65%
Localisation	72%
Entire Engine	60%

Table 6.2: Function coverage percentages for various systems in the engine.

6.3 Coverage

Due to time constraints, we did not develop a substantial number of structural tests. Through functional tests we have verified that individual engine features function as we would expect, but we do not know how much of the actual engine code is covered by these tests. Therefore, we used a coverage analysis tool while running all of our functional tests to determine the percentage of functions that are executed. Generally one would use coverage analysis during structural testing, but in our case our most complete and frequently used tests were functional tests. The results of the coverage analysis is given in Table 6.2. These numbers are all relatively low. One would expect a fully implemented test plan to cover more than 90% of functions, since one generally cannot have any confidence in functions that are never called during a testing phase. There are several reasons why our function coverage percentages are low. The first is the fact that our tests are functional and not structural. Hence, the focus of our tests was not to exhaustively test every execution path. Secondly, upon a more in-depth analysis of the coverage results, we found that many of the uncalled functions are indeed overloaded functions and alternate constructors provided for convenience. Furthermore, the functional tests did not induce many errors or exceptions in the engine, and hence many functions related to error handling are also not called. On average a class in the engine has six uncalled functions. It is thus apparent that the engine requires more extensive testing in the future, especially as far as structural testing is concerned. Furthermore, this also leads one to consider whether all of the overloaded and convenience functions are in fact necessary. One could possibly use coverage data to identify certain superfluous functions that could be removed.

In this chapter we discussed testing measures utilised during the engine implementation. Our tests were mainly functional tests. These tests enabled us to identify which engine features work as expected and which contained problems that we then ultimately corrected. However, we did not include many structural tests in our test plan. The lack of structural tests is apparent through the results of the coverage analysis in this section. We have reasonable confidence in the results of our functional tests, but the engine should

be tested more thoroughly in the future.

In the next chapter we discuss the results and conclusions of this project.

Chapter 7

Results and Conclusion

We set out to design and implement a generic 3D virtual environment platform with specific goals in mind, and we completed the project successfully.

The goals for the platform were:

- to be able to manage and render 3D virtual environments,
- to be adaptable and extendable for specific needs,
- to have longevity: it should remain useful in the future,
- to be well documented,
- to have a generic input management system,
- to have a customisable graphical user interface (GUI) system,
- to include a replay system,
- to have a data capturing system,
- to provide a remote monitoring facility where therapists can monitor a session over a network,
- to provide multi-lingual support in games and therapy tools through a language localisation system, and
- to provide a framework upon which tools can be built to aid non-programmers in constructing educational games and ASD therapy tools.

We built our engine using OGRE as the primary library, since the open-source game engines we surveyed did not satisfy all of the requirements set out in our project goals.

Using OGRE, along with the numerous plug-ins and add-ons available for it, we had more control over the design of the engine and its API. Examples of two systems included in our engine through OGRE add-ons are GUI support using MyGUI with a WYSIWYG layout editor and physics support through NxOGRE and Nvidia PhysX.

We designed and implemented a task management system along with an internal communication system. This allows the engine to effectively manage multiple subsystems. Tasks included in our implementation are for rendering, audio, GUI management, input, networking and a physics system. The communication system allows these tasks to communicate and also allows user applications and plug-ins to communicate with systems in the engine. Messages sent through the communication system are contained in a class hierarchy with an abstract base class, namely the Message class. Hence, new messages can be defined in the future by extending the Message class hierarchy.

Our engine includes methods for managing game objects, since game engines typically have to manage numerous game objects that a user can see or interact with, such as tables and chairs in the game world. We created a hierarchy of classes to represent these game objects, with an abstract base class, namely the GameObject class. Classes in this hierarchy include support for rendering by OGRE as well as support for physics and collision detection. Furthermore, we created a system through which the creation and destruction of instances of these GameObject classes could be managed. This was accomplished by using an abstract factory, namely the GameObjectFactory.

A plug-in framework, allowing developers to extend engine functionality through plug-ins instead of modifying the engine code directly, was implemented. This plug-in system has proven successful as we have used it to add support for Java (the Lua scripting interface is directly integrated with the engine) and to create an audio plug-in for the engine based on the OgreOggSound add-on for OGRE.

Throughout the development of the engine we maintained extensive documentation of the engine API. What is lacking, however, is introductory tutorials on the basic usage of the engine. This will be added in the future.

We implemented a hardware abstraction layer for the input management system of the engine. This makes it possible to add support for new types of input peripherals in the future, while keeping the manner in which input is reported to games or therapy tools that utilise the input system reasonably generic. Such a hardware abstraction layer is important for this project since some individuals with ASD suffer from motor difficulties [26, 27], and may require specialised input peripherals. We have only implemented support for standard input peripherals at this stage.

Using the MyGUI add-on for OGRE our platform supports customisable GUIs through the layout editor of MyGUI and its customisable appearance files.

We successfully designed and implemented a replay system in the engine. The replay system records the various messages sent through the engine as a consequence of user actions. When the replay is played back, the saved messages are transmitted within the correct time intervals. This system was successfully tested with the social skills therapy tool discussed in Section 5.3. However, from our tests in Chapter 6, the system does suffer some synchronisation issues when the engine runs at an excessively low frame rate. Hence, in the future, synchronisation could be added to the replay system by periodically saving the game state to the replay file along with the messages to always ensure correct playback in these situations. Inclusion of a replay system was specifically requested by therapists, as it would allow them to evaluate the progress that individuals have made through the use of therapy tools.

The engine includes a generic data capturing framework. We successfully implemented an SQLite-based version of this framework. This implementation is used in all three test applications discussed in Chapter 5. The data capturing system saves the occurrence of certain in-game events, answers provided to questions or in-game decisions the player has to make. The data capturing system works well and teachers were positive about the information that it provided during the test phase of the Journey to the Moon game discussed in Section 5.2.

The engine includes a network component with configurable behaviours. These behaviours determine how the system responds to messages that are received over the network and what information is transmitted to hosts. We implemented a basic peer-to-peer network model. Similar to the replay system, this network model sends engine messages about relevant user actions to all connected peers on the network. Every peer maintains its own state of the game. This network model was successfully tested in the social skills therapy tool, discussed in chapters 5 and 6. However, during testing we did find that synchronisation issues do occur with this network model over a period of time. This occurs since there is currently no automatic synchronisation mechanism included in the peer-to-peer model. This is left as future work and possible approaches are discussed in Section 3.15. We did, however, successfully implement basic synchronisation support specifically for the social skills tool by periodically sending the position of the player avatar to monitoring peers. The network components were specifically included due to requests from therapists, as it would allow them to monitor the progress of individuals using therapy tools.

We implemented a language localisation component for the engine. The multi-lingual context of South Africa necessitated the inclusion of such a component. This makes it possible for games to provide text output in multiple languages. All of the games and tools discussed in Chapter 5 included both Afrikaans and English language files.

We successfully implemented scripting interfaces for the engine for Lua and Java. The

Lua interface was used to implement all of the games and tools in Chapter 5.

One of the goals of this project is to include some framework in the engine that can be used by non-programmers to develop tools for easily constructing educational games and therapy tools. We believe the general functionality provided by the engine along with the scripting interface can be used to build such tools. A tool can, for instance, allow individuals to create a game through some visual programming interface and then output a script. Our engine can be used to run the script in order to play the game. This does, however, require substantial work from developers to first develop such construction and script generation tools.

The engine and the Journey to the Moon game were tested at a local school for a period of 20 days during which no failures or errors occurred. Feedback from the teacher involved in the test is included as Appendix B and is mainly positive.

Currently there are no automatic synchronisation methods implemented for replays or for the networking component. This could be added in the future to improve the overall quality and correctness of these components.

From the discussion on testing in Chapter 6 it is important that the engine be tested more thoroughly in the future. A specific focus on structural tests where we try and maximise the test coverage of the engine would be advantageous.

In the future researchers may wish to incorporate virtual agents into games or therapy tools created with the engine. Currently the engine does not have support for path finding, which is important if the virtual agents need to move around in the game world. Furthermore, virtual agents would also need to sense their surroundings. It is important to ensure that our engine supports ways in which virtual agents can determine what objects are around them in the virtual world. This could be done through collision detection, where virtual agents have view geometries and we test for intersection with their view geometries and objects in the virtual world. PhysX supports such collision detection, but we must ensure that these features are easily accessible to developers through our engine API.

In conclusion, we set out to design and implement a platform for the development of 3D virtual environments for use in educational games and ASD therapy tools. Our test implementations in Chapter 5 as well as Chapter 6 lead us to believe that we have achieved the goal for this project. The engine manages 3D virtual worlds and is adaptable and extendable through the use of object-oriented design paradigms, the task management and communication interface as well as the plug-in system. Specific functionality that is useful for educational games and therapy tools include the data capturing system, the replay system and remote monitoring through the networking system. The engine also makes it possible to implement multi-lingual games and tools through the language localisation

component. We have therefore, in our opinion, succeeded in developing a generic and extendable 3D virtual environment platform for the development of educational games and ASD therapy tools. We successfully developed several game and tool implementations to test the system. Thus, we have achieved our thesis goal.

Appendices

Appendix A

Design Patterns

In this appendix we discuss a number of design patterns that are mentioned throughout the thesis. The first pattern we discuss is that of a singleton.

A.1 Singletons

We provide a brief summary of singletons directly from [25]. A singleton is a member of the creational family of design patterns. As the name implies, this family of patterns provides guidelines or limitations on the manner in which an instance is created. The intent of a singleton is to "ensure a class only has one instance, and provide a global point of access to it". The singleton pattern should be used when:

- there must be exactly one instance of a class, and there must be a global access point to it, and
- when the single instance should be extensable by sub-classing, where clients should be able to use an extended instance without modifying their code.

We define a singleton in definition A.1.1.

Definition A.1.1 (Singleton) *A class is a singleton if and only if the following is true:*

- *mechanisms are in place to ensure that there can at most one instance of the class at any point in time, and*
- *there is a global, well-known, access point to the instance.*

In the next section we discuss a design pattern that is useful for managing the creation of instances of classes from a class hierarchy with an abstract base class.

A.2 Abstract Factory

This summary of abstract factories are again taken from [25]. The abstract factory is a member of the creational family of design patterns. The intent of this pattern is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes."

Abstract factories should be used in the following situations:

1. if a system should be independent of the manner in which its objects are created,
2. if a system could be configured with one of a number of families of objects,
3. if a family of related objects is designed to be used in conjunction with one another, and this constraint must be enforced, and
4. if one wants to provide a library of classes by only revealing their interfaces and not their implementations.

Definition A.2.1 (Abstract Factory) *A class is an abstract factory if and only if it provides an interface for the creation of families of dependent objects where the concrete class of the object to be created is not specified.*

In the next section we discuss classes that can be used to wrap function calls with.

A.3 Functor

A functor, also called a function object, is an object that encapsulates a call to a function. Typically a functor would also support the normal function call syntax to actually call its wrapped function [65]. Functors are widely used for callback functions. It allows one to hide the detail of the function being called.

Appendix B

Educational Game Questionnaire

1. Please state the grade of each child who used the software.

Child	1	2	3	4	5	6	7	8	9	10
Grade	1	1	1	1	1	1	1	1	1	1

2. For each child, over how long a period did they use the software?

Child	1	2	3	4	5	6	7	8	9	10
Days	20	20	20	20	20	20	20	20	20	20

3. For each child, how many times did they use the software over the whole period?

Child	1	2	3	4	5	6	7	8	9	10
Total number of sessions	20	20	20	20	20	20	20	20	20	20

4. For each child, how regularly (how many times per week) did they use the software over the whole period?

Child	1	2	3	4	5	6	7	8	9	10
Regularity of sessions	5	5	5	5	5	5	5	5	5	5

5. For each child, did they like using the software? (very much, neutral, disliked)

Child	Popularity
1	Very much
2	Very much
3	Very much
4	Very much
5	Very much
6	Very much
7	Very much
8	Very much
9	Very much
10	Very much

6. For each child, what aspect of the software did they like best?

Child	Aspect of software liked best
1	Space ship
2	Space ship
3	Maths
4	Maths
5	Sounds
6	Sounds
7	Space ship
8	Faster thinking maths
9	Sounds
10	Space ship

7. For each child, what aspect of the software did they like the least?

Child	Aspect of software liked least
1	Maths
2	Maths
3	None
4	None
5	None
6	None
7	None
8	None
9	None
10	None

8. Did any of the children find it stressful to use the software?

Child	1	2	3	4	5	6	7	8	9	10
Stressful?	No	No	No	No	No	No	No	No	No	No

9. For those children that found it stressful to use the software, did their anxiety decrease with repeated sessions, or not? Answer *yes*, *no*, *not applicable (NA)*.

Child	1	2	3	4	5	6	7	8	9	10
Decrease in stress	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA

10. In your opinion, did the use of the software improve the educational skills of the children who used it regularly? Answer *yes* or *no*.

Child	1	2	3	4	5	6	7	8	9	10
Improvement	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No

11. In your opinion, would it be beneficial for the children to also use the software without supervision, to practise certain skills? Answer *yes* or *no*.

Child	1	2	3	4	5	6	7	8	9	10
Without supervision	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

12. In your opinion, are there any advantages to using the software as an educational tool, as opposed to not having the software in the classroom? If yes, state the advantages.

Advantages?	Yes
Advantage 1	Children with fine motor skills (handwriting) will benefit.
Advantage 2	It helps with anxiety because it is more informal.
Advantage 3	They find it interesting and with ADD/ADHD children, they focus very well.

13. For each child, please indicate whether they had difficulty using the keyboard (answer *yes* or *no*).

Child	1	2	3	4	5	6	7	8	9	10
Keyboard difficulty	No	No	No	No	No	No	No	No	No	No

14. For each child, please indicate whether they had difficulty using the mouse (answer *yes* or *no*).

Child	1	2	3	4	5	6	7	8	9	10
Mouse difficulty	No	No	No	No	No	No	No	No	Yes	Yes

15. If you answered yes for any child in 13 or 14 above, please state in more detail what the difficulty was, and how you think it can be solved.

Difficulty	Fine motor skills problems.
Possible solution	Bigger mouse or touch screen.

16. For each child, please indicate whether they had difficulty comprehending the screen layout (answer *yes* or *no*).

Child	1	2	3	4	5	6	7	8	9	10
Screen layout	No	No	No	No	No	No	No	No	No	No

17. For each child, please indicate whether the elements on the screen were too bright/flashy/distracting (answer *yes* or *no*).

Child	1	2	3	4	5	6	7	8	9	10
Elements on screen distracting	No	No	No	No	No	No	No	No	No	No

18. For each child, please indicate whether they found it difficult to keep focused while using the software (answer *yes* or *no*).

Child	1	2	3	4	5	6	7	8	9	10
Focus	No	No	No	No	No	No	No	No	No	No

19. If you answered *yes* for any child in 18 above, please state in more detail what the difficulty was, and how you think it can be solved.

Difficulty	NA
Possible solution	NA

20. For each child, please state whether sound effects are advisable in the software.

Child	1	2	3	4	5	6	7	8	9	10
Sound	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

21. Apart from Afrikaans and English, would it be helpful to have the same software in other languages? If so, please state the languages.

Languages	No

22. Are there any specific colour combinations that you can recommend for the graphical user interface?

Colours	Anything calm, blue, baby colour, grey etc.

23. In your opinion, is the size of the buttons and icons on the screen appropriate?

Size	Could be a little bigger.

24. Should buttons have pictures only, text only, or pictures and text?

Pictures		**Text**		**Pictures and text**	✓

25. Please write down any additional comments you may have on the usability of the software.

Usability	It was very user friendly and the children could operate it independently.

26. Please write down any additional comments you may have on the educational aspects of the software.

Educational aspects	ADD/ADHD children stayed focused and they enjoyed it a lot. They had to think fast (they enjoyed it) with one of the maths games. They were also forced to work independently and read words associated with sounds. They loved it!

Bibliography

[1] Extensible Markup Language (XML), last accessed in August 2008. http://www.w3.org/XML/.

[2] Eclipse, last accessed in August 2009. http://www.eclipse.org/.

[3] Netbeans, last accessed in August 2009. http://www.netbeans.org/.

[4] Bullet Physics Library, last accessed in July 2009. http://www.bulletphysics.com.

[5] Deck13 Interactive, last accessed in July 2009. http://www.deck13.com/english/index.html.

[6] Runic Games, last accessed in July 2009. http://www.runicgames.com/.

[7] Anderson, E. F., Engel, S., Comninos, P., and McLoughlin, L. The Case for Research in Game Engine Architecture. In *Future Play '08: Proceedings of the 2008 Conference on Future Play*, pages 228–231. ACM, 2008.

[8] Autodesk, Inc. 3D Studio Max, last accessed in July 2009. http://usa.autodesk.com/adsk/servlet/pc/index?id=13567410&siteID=123112.

[9] Baron-Cohen, S., Scott, F., Allison, C., Williams, J., Bolton, P., Matthews, F., and Brayne, C. Prevalence of Autism-Spectrum Conditions: UK School-based Population Study. *The British Journal of Psychiatry*, 194(6):500, 2009.

[10] Beazley, D., Fulton, W., Betts, O., Lenz, J., Gossage, M., and Wang, J. Simplified Wrapper and Interface Generator, last accessed in March 2009. http://www.swig.org.

[11] Brand, E. Private Communication, 28 May 2009.

[12] Castaneda, P. Object Oriented Input System, last accessed in July 2008. http://sourceforge.net/projects/wgois.

[13] Charman, T. The Prevalence of Autism Spectrum Disorders. *European child & adolescent psychiatry*, 11(6):249–256, 2002.

[14] Cheng, Y., Moore, D., McGrath, P., and Fan, Y. Collaborative Virtual Environment Technology for People with Autism. In *ICALT '05: Proceedings of the Fifth IEEE International Conference on Advanced Learning Technologies*, pages 247–248. IEEE Computer Society, 2005.

[15] Cobb, S. V. G., Neale, H. R., and Reynolds, H. Evaluation of Virtual Learning Environments. In *Proceedings of the 2nd European Conference on Disability, Virtual Reality and Associated Technologies, Skövde, Sweden*, pages 17–23. 1998.

[16] Creative Labs. OpenAL, last accessed in July 2009. `http://connect.creativelabs.com/openal/default.aspx`.

[17] Dalmau, D. S. *Core Techniques and Algorithms in Game Programming*. New Riders Publishing, 2003.

[18] DeLoura, M. *Game Programming Gems 2*. Charles River Media, Inc., Hingham, MA, 2001.

[19] Demichelis, A. Squirrel, last accessed in April 2008. `http://squirrel-lang.org/`.

[20] Epic Games. The Unreal 3 Engine, last accessed in July 2009. `http://www.unrealtechnology.com/`.

[21] Evmenov, G. MyGUI, last accessed in July 2009. `http://sourceforge.net/projects/my-gui/`.

[22] Fine, R. The Enginuity Engine, last accessed in March 2008. `http://www.gamedev.net/reference/programming/features/enginuity1/`.

[23] Forrester, J. Private Communication, 24 April 2008.

[24] Gagne, G., Galvin, P., and Silberschatz, A. *Operating System Concepts*. Wiley, 2005.

[25] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Indianapolis, IN, 2007.

[26] Ghaziuddin, M. and Butler, E. Clumsiness in Autism and Asperger Syndrome: a Further Report. *Journal of Intellectual Disability Research*, 42:43–48, 1998.

[27] Ghaziuddin, M., Butler, E., Tsai, L., and Ghaziuddin, N. Is Clumsiness a Marker for Asperger Syndrome? *Journal of Intellectual Disability Research*, 38:519–527, 1994.

[28] Gist, M. Cross Thread Communication and Multi-Threaded Loading, last accessed in March 2009. `http://code.google.com/soc/2008/crystal/about.html`.

[29] GNU. Gettext, last accessed in March 2009. `http://www.gnu.org/software/gettext/`.

[30] Id Software. Quake 3 Source Code (GPL), last accessed in July 2008. `ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip`.

[31] Ierusalimschy, R., De Figueiredo, L., and Celes, W. *Lua 5.1 Reference Manual*. Lua.Org, 2006.

[32] Junker, G. *Pro OGRE 3D Programming*. Apress, 2006.

[33] Kaner, C., Falk, J., and Nguyen, H. *Testing Computer Software*. John Wiley & Sons, Inc., 1999.

[34] Kerr, S. J., Neale, H. R., and Cobb, S. V. G. Virtual Environments for Social Skills Training: the Importance of Scaffolding in Practice. In *Proceedings of the Fifth International ACM Conference on Assistive Technologies, Edinburgh, Scotland*, pages 104–110. ACM, 2002.

[35] Kilgard, M. GLUT – The OpenGL Utility Toolkit, last accessed in March 2009. `http://www.opengl.org/resources/libraries/glut/`.

[36] Lazar, J. *Universal Usability*. Wiley, 2007.

[37] Linden Labs. Second Life, last accessed in March 2009. `http://secondlife.com/`.

[38] Manzur, A. toLua++, last accessed in March 2009. `http://www.codenix.com/~tolua/`.

[39] Moore, D., McGrath, P., and Thorpe, J. Computer-Aided Learning for People with Autism – a Framework for Research and Development. *Innovations in Education and Teaching International*, 37:218–228, 2000.

[40] NVIDIA Corporation. NVIDIA PhysX Physics Simulation for Developers, last accessed in July 2009. `http://developer.nvidia.com/object/physx.html`.

[41] Obiltschnig, G., Fabijanic, A., and Schojer, P. Portable Components C++ Libraries, last accessed in July 2009. `http://pocoproject.org`.

[42] Ousterhout, J. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, 1998.

[43] Parsons, S. and Mitchell, P. The Potential of Virtual Reality in Social Skills Training for People with Autistic Spectrum Disorders. *Journal of Intellectual Disability Research*, 46:430–443, 2002.

[44] Parsons, S., Mitchell, P., and Leonard, A. Using Virtual Environments for Teaching Social Understanding to 6 Adolescents with Autistic Spectrum Disorders. *Journal of Autism and Developmental Disorders*, 37(3):589–600, 2007.

[45] Pellegrino, J. D. and Dovrolis, C. Bandwidth Requirement and State Consistency in Three Multiplayer Game Architectures. In *NetGames '03: Proceedings of the 2nd Workshop on Network and System Support for Games*, pages 52–59. ACM, 2003.

[46] Perry, W. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., 2000.

[47] Peterson, L. and Davie, B. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 2007.

[48] Rasterbar Software. Luabind, last accessed in March 2009. `http://www.rasterbar.com/products/luabind.html`.

[49] Rob, P., Coronel, C., and Crockett, K. *Database Systems: Design, Implementation and Management*. Thomson Learning, 2004.

[50] Schmidt, C. and Schmidt, M. Three-Dimensional Virtual Learning Environments for Mediating Social Skills Acquisition Among Individuals with Autism Spectrum Disorders. In *IDC '08: Proceedings of the 7th International Conference on Interaction Design and Children*, pages 85–88. ACM, 2008.

[51] Sherrod, A. *Ultimate 3D Game Engine Design & Architecture*. Charles River Media, Inc., 2006.

[52] Smed, J., Kaukoranta, T., and Hakonen, H. A Review on Networking and Multiplayer Computer Games. In *Multiplayer Computer Games, Proceedings of the International Conference on Application and Development of Computer Games in the 21st Century*, pages 1–5. 2002.

[53] Streeting, S. Object Oriented Graphics Engine, last accessed in July 2009. `http://www.ogre3d.org`.

[54] Streeting, S. Object Oriented Graphics Engine Featured Projects, last accessed in July 2009. `http://www.ogre3d.org/gallery/`.

[55] Strickland, D. Virtual Reality for the Treatment of Autism. *Studies in Health Technology and Informatics*, 44:81–86, 1997.

[56] Thayer, A. and Kolko, B. Localization of Digital Games: The Process of Blending for the Global Games Market. *Technical Communication, Washington*, 51(4):477–488, 2004.

[57] The Blender Community. Blender, last accessed in July 2009. `http://www.blender.org`.

[58] The Crystal Space Community. Crystal Space 3D, last accessed in March 2009. `http://www.crystalspace3d.org`.

[59] The OGE Community. The Open Game Engine, last accessed in April 2008. `http://www.opengameengine.org`.

[60] The OGE Community. The Open Game Engine Design Overview, last accessed in August 2009. `http://sourceforge.net/apps/mediawiki/oge/index.php?title=Engine_Overview_and_Object_System_Discussion_Summary`.

[61] The Panda3D Development Team. Panda3D, last accessed in April 2008. `http://www.panda3d.org`.

[62] Tulip, J., Bekkema, J., and Nesbitt, K. Multi-Threaded Game Engine Design. In *Proceedings of the 3rd Australasian Conference on Interactive Entertainment*, pages 9–14. Murdoch University, Australia, 2006.

[63] Turner, P. Crazy Eddie's GUI System, last accessed in March 2008. `http://www.cegui.org.uk`.

[64] Valve Software. The Source Engine, last accessed in July 2009. `http://source.valvesoftware.com/`.

[65] Vandevoorde, D. and Josuttis, N. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2003.

[66] Vermeulen, P. *I Am Special: Introducing Children and Young People to Their Autistic Spectrum Disorder*. Jessica Kingsley Publishers, 2000.

[67] Wright, S. and Tischer, S. Architectural Considerations in Online Game Services Over DSL Networks. In *Proc. IEEE International Conference on Communications-(ICC'04), IEEE Communications Society, Paris, France*, pages 1380–1385. 2004.