A GENERIC DEVELOPMENT PLATFORM FOR ASD THERAPY TOOLS

Lynette van Zijl and Morné Chamberlain
Department of Computer Science, Stellenbosch University, Stellenbosch, South Africa

Keywords: Autism, Assistive technologies, Computerised therapy tools, Virtual environments, Game engines, Games.

Abstract: We discuss the design and implementation of a generic development platform for 3D virtual environments for autism spectrum disorder therapy tools. The platform is intended to enable researchers and therapists alike to quickly develop individual therapy software without having to resort to extensive software development. The platform includes specific features requested by therapists, such as replays, data capturing and remote monitoring over a network. We also discuss the implementation of a number of different therapy tools, and their subsequent evaluation at local schools.

1 INTRODUCTION

Over the past two decades, there has been a worldwide marked increase in the occurrence of autism spectrum disorders (ASDs) in children. Assistance to these children requires highly-trained professionals, and intensive one-on-one therapies. The accompanying expense is often prohibitive for lower-income parents, and government-supported schools have exceptionally long waiting lists. This work aims to alleviate the problem by providing a framework to fast-track the development of computerised ASD therapy tools.

ASDs are a complex range of disorders, characterised primarily by a lack of communication, language and social skills. In particular, children with ASD usually interpret language literally, and find body language and social innuendo extremely difficult to understand. This immediately means that any therapies are language and culture dependent, so that computerised therapy tools from America and Europe can seldom simply be re-used in the South African context. There is thus an immediate need for South African-specific computerised therapy tools.

In addition to the need for therapy tools, researchers investigating ASDs from a medical or psychological perspective often require specific software to interact with children with ASDs. A generic platform for the development of computerised ASD tools can be used in this context as well. The advantage would be that research groups do not have to spend time on implementation of graphical environments and avatars, but can concentrate on the specific tools directly related to their research.

We present in this article the design and implementation of a generic platform for the development of computerised ASD therapy and research tools, with the specific aim to provide free, culture-independent software, on a generic platform. As additional specifications, we require our platform to be network-capable, multi-threaded, have low resource requirements, and provide a middleware interface for subsequent developments.

The rest of this article is organised as follows: we give a brief overview of related work in section 2, followed by a detailed exposition of our design in section 3. We discuss the implemented therapy tools in section 4, and conclude in section 6.

2 RELATED WORK

One of our motivations for the development of computerised ASD therapy tools is the premise that computerised practise relieves much of the stress caused by the human interaction necessary in therapy sessions (see for example (Goldsmith and Leblanc, 2004)). However, this is a valid starting point only if computerised therapy tools would indeed have beneficial results. This had been shown to be the case, as we briefly point out below.

van Zijl L. and Chamberlain M. (2010).
A GENERIC DEVELOPMENT PLATFORM FOR ASD THERAPY TOOLS.
In *Proceedings of the 2nd International Conference on Computer Supported Education*, pages 82-89
DOI: 10.5220/0002793800820089
Copyright © SciTePress

2.1 Computerised ASD Therapy Tools

The use of virtual environments as teaching tools for individuals with and without special needs has been extensively researched – see for example (Cobb et al., 1998). General consensus is that virtual environments can be advantageous as teaching tools, given careful design of the interface and sound underlying educational principles (Kerr et al., 2002). A number of studies had also been undertaken to investigate the use of virtual environment teaching tools specifically for individuals with ASDs (Moore et al., 2000; Parsons and Mitchell, 2002; Parsons et al., 2007; Strickland, 1997). Again, these studies conclude that virtual environment teaching tools are advantageous given careful attention to the design.

For example, (Parsons et al., 2007) built two virtual social scenarios, namely, a bus and a cafeteria. The scenarios were both goal-based. Participants had to negotiate a social environment to reach the goal of, respectively, boarding the bus and obtaining food. The study showed that structured, goal-oriented VEs enabled participants to repeatedly practise skills. Moreover, these skills were transferred to real life situations with relative ease.

Given supporting evidence that VEs can be useful in ASD therapy, we next consider the technical challenges faced in the development of such a VE. As computer game engines typically contain all the components required in such a VE, we consider the design of some existing game engines.

2.2 Existing Game Engines

In our development of a generic VE, we could either use existing software, or create our own. Since there are already many open-source game engines available, our initial approach was to find a suitable engine, and simply build the rest of the project on such an engine. We therefore evaluate some existing game engines in this section.

The Open Game Engine (OGE) is an open-source community project that strives to create a powerful 3D computer game engine (The OGE Community, 2008). It has several features that we require, such as high-level scripting language support. OGE makes extensive use of proven open-source projects, such as the Object-oriented Graphics Rendering Engine (OGRE) and Open Input System (OIS). The use of OGE for our research initially seemed promising. However, at the time of writing the engine is not stable, and is undergoing a complete redesign.

Panda3D (The Panda3D Development Team, 2008) is a 3D game engine originally developed by *Walt Disney Entertainment*, but is currently maintained at Carnegie Mellon University. Panda3D is an extensive 3D game engine that has been used in commercial projects, such as *Pirates of the Caribbean Online*.

The Panda3D engine has many desirable features, including scene graphs, a task oriented structure, an audio system, collision detection and a physics engine. The Panda3D engine is programmed in C++, but the engine has a full Python interface that is automatically generated from the C++ interface. While Panda3D is a full-featured engine with excellent functionality, we found that it would be difficult to extend. Also, it is a large and complicated system that would be difficult to install on older computers. Lastly, the documentation on the engine was lacking.

The *Enginuity Engine* (EE) (Fine, 2008) is an open-source engine implemented in C++. The design and development of the engine is well-documented.

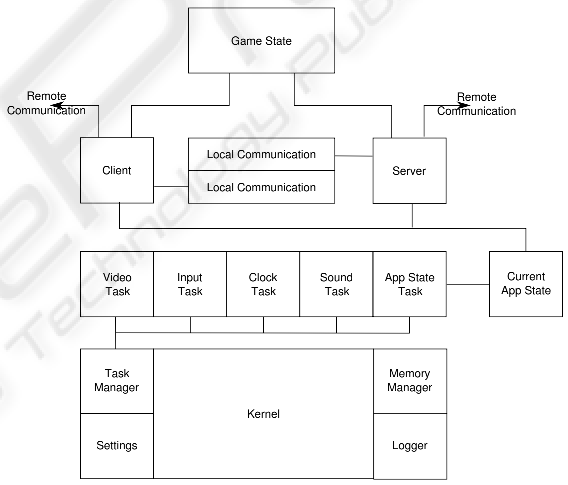


Figure 1: Design diagram of the *Enginuity Engine.*

The design of the EE is outlined in figure 1, taken from (Fine, 2008). The design is focused 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 execution of tasks. A list of common tasks is given in figure 1. The order of task execution is determined by the priority of each task. The EE uses a direct communication model (tasks call functions from other tasks directly) and favours a single-threaded engine design.

The design of the EE satisfies many of the requirements for our virtual environment engine as it is generic and easy to extend or adapt for specific needs. However, the EE is intended as a technical example of a design for a good game engine, and not as a complete engine itself. For example, it only includes limited graphics rendering capabilities, it would require extensive adaptation to link to libraries like OIS and OGRE, and it is single-threaded (see our discussion in section 3 on the advantages of multi-threading).

Crystal Space (The Crystal Space Community, 2009) is a cross-platform graphics and game engine. It is a mature library with many features. Crystal Space can be extended by plug-ins. Some notable plug-ins that have been implemented for Crystal Space include an audio plug-in and a physics engine plug-in using the Open Dynamics Engine. The engine also has a number of wrapper libraries that enables its use from scripting languages such as Python. Crystal Space does not have built-in functionality for replays and data capturing. Furthermore, its existing networking support is not at the level that we require.

The Object-oriented Graphics Rendering Engine (Streeting et al., 2009; Junker, 2006) is a cross-platform open-source graphics rendering engine. It is a mature library that has been in development for a number of years. It was designed to have a simple API. Although OGRE is not a full game engine, it has an active community that has developed a number of high-quality add-ons. Examples of these add-ons include MyGUI, a graphical user interface add-on, NxOGRE, an Nvidia PhysX based physics engine add-on and OgreOggSound, an OpenAL (Creative Labs, 2009) based audio add-on. Therefore, one could use OGRE and a number of its add-ons to provide a major portion of the basic functionality of a game engine. The advantage of this would be the fact that it would offer more control over the interface and the size of the ultimate platform than if we were to use a complete game engine.

In recent years the online social networking phenomenon appeared. An example of such a system is Second Life (Linden Labs, 2009). Second Life allows users to connect to an online virtual world. Users are represented by avatars and can interact with other users in the virtual world. The Second Life client is available as open-source, and thus it should be possible to extend the client to suit our needs. However, the cost of bandwidth in South Africa makes the use of an online system prohibitive. Furthermore, in the case of ASD therapy tools, we require a level of control over the virtual world that would not be attainable in an online solution.

We therefore opted to build our platform with OGRE and its add-ons as the base. We largely base the design of our platform on the design of the EE, but with some of the desirable features from OGE, Panda3D and Crystal Space included.

In the next section we shift the focus of the discussion to the design and implementation of our engine and virtual environment platform.

3 DESIGN

As stated above, our engine uses OGRE and a number of its add-ons as a base. The remainder of the engine design is based strongly on the *Enginuity Engine* (Fine, 2008), with the notable exception that our engine is multi-threaded. This implies that different modules and classes in the engine communicate using a message-passing system, as opposed to the EE which uses direct communication. Multi-threading allows for different threads to execute simultaneously and on different cores. This allows for better synchronization between, for example, video and sound, and in general improves the reaction times in the therapy tools.

An engine such as ours is responsible for managing various types of activities. These activities include: management of input received from input peripheral devices, playing audio, executing scripts and game logic and rendering graphics. In order not to reinvent the wheel, we opted to use a number of proven third party libraries to manage some of the above mentioned activities. These include OIS for managing input and Nvidia PhysX for physics and collision detection.

In order for the engine to effectively perform its duties, and remain reasonably generic and extensible, we opted to use a design paradigm similar to those found in an operating system (Gagne et al., 2005), namely, using a kernel and a set of tasks. This kernel and task paradigm is also used and promoted in the EE (Fine, 2008) (see section 2).

The core of the engine consists of a kernel and a set of tasks. Tasks are divided into two categories, namely, sequential tasks and concurrent tasks. The kernel is responsible for managing the execution of each task. Sequential tasks are executed in a round-robin fashion by the kernel and in the same thread as the kernel. Each concurrent task has its own thread and is executed in parallel with the kernel thread.

Some common tasks include:

Timer task: keeps track of the amount of time that has passed during engine execution and between rendered frames.

Input task: manages the various peripheral input devices.

Video task: the video task is responsible for rendering the virtual environment to the screen.

Tasks communicate via a message-passing system, primarily to avoid synchronisation issues with direct communication. The internal communication model of the engine is discussed in section 3.1.

In order to control a task, or send messages between tasks, we require a management application programming interface (API). Thus most tasks have an associated manager class that is used to control tasks. An example would be a manager class that sends a message to the video task, instructing it to change the screen resolution. Certain classes that are not tasks, but also require some form of management, also have manager classes (the kernel is an example of such a class). The kernel manager provides functions for adding, removing, suspending and resuming tasks in the kernel.

In figure 2 we give an overview of the design of our engine with respect to the elements we have discussed. In the following sections we discuss these elements in more detail.

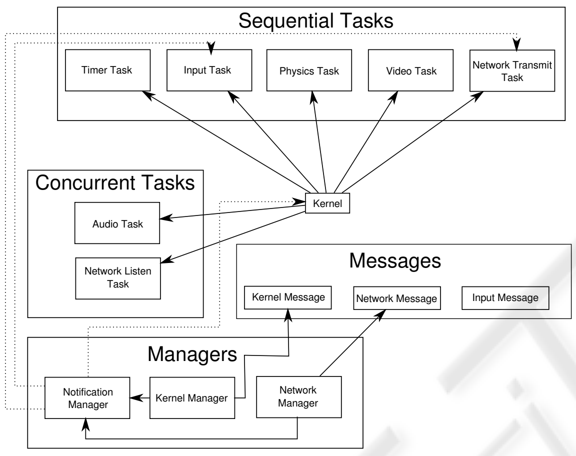


Figure 2: Design overview of our engine.

3.1 Internal Communication

In the *Enginuity Engine* (Fine, 2008), a direct communication model is used to facilitate communication between the various systems and tasks. Although this approach is simple to implement, it provides no rules or API to follow when new systems are added to the engine. Also, with the introduction of concurrent tasks in multiple threads in our engine, it became essential to have a more robust form of communication.

The message-passing system we designed consists of a notification manager class, various message queues and various message types. Objects must register to receive messages. When registering, the object must provide the type of messages it wishes to receive and provide a function that is to be called when such a message is received. Objects can register to receive any number of message types.

Messages can be delivered synchronously or asynchronously. In the synchronous case, the message handler function for the object is called as soon as a synchronous message of that type is sent. In the asynchronous case, all sent messages are queued independently at each object that has registered to receive messages of that type. It is then the responsibility of the receiving object to periodically call a function to process its queue of messages. For each message waiting in the queue, the message handler function that the object provided when registering to receive messages, is called.

Messages of different types are implemented as classes. Message classes can extend one another. The only requirement is that all message classes have the generic Message class as their base class.

3.2 The Game Object Model

The game object model is a hierarchy of classes, starting with a generic and extensible base class, that can be used to construct any in-game object. We provide a basic set of generic classes in the hierarchy. If the basic functionality does not satisfy the requirements for a specific situation, new classes can be derived from the existing ones. Figure 3 illustrates the hierarchy.

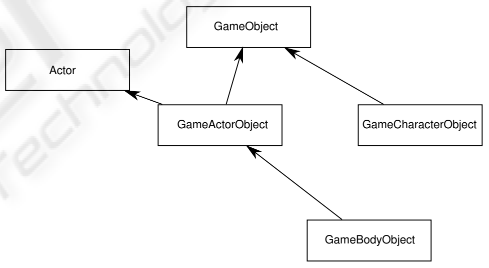


Figure 3: The hierarchy of classes that represent the various types of objects in the virtual world.

The base class of the hierarchy is the `GameObject` class. The `GameObject` is the most basic class that can be used to construct representable virtual objects, such as tables and chairs, for the virtual world. The `GameObject` does not provide any mechanisms to allow for visualisation. The `GameBodyObject` is the first class in the hierarchy to add such functionality.

The `GameActorObject` extends the `GameObject` class to include functionality that is associated with a so-called actor. The concept of an actor comes from the underlying physics system that we use, namely, the Nvidia PhysX engine. The `Actor` class that is visible in the hierarchy is merely a wrapper class that implements the functionality of a PhysX actor in our engine. Actors in the virtual world can be described as those objects that have the following properties: a position, an orientation, a velocity, shape, and mass. Actors do not have any visualisation associated with them: one cannot see an actor on its own. A body is simply an extension of an actor and includes some form of visualisation.

Therefore, the `GameActorObject` class extends the `GameObject` class and the `Actor` implementation from the physics engine. The `GameBodyObject` class then extends the `GameActorObject` class and adds the required functionality (support for setting a 3D model and textures) so that the object can be rendered by the 3D engine.

During execution of a game or tool implemented with the engine, many class instances derived from the `GameObject` class will be created and destroyed. In order to manage the creation of these different `GameObjects`, we use an abstract object factory.

3.3 Abstract Object Factories

An abstract factory is: "a class that provides an interface for the creation of families of dependent objects where the concrete class of the object to be created is not specified" (Gamma et al., 2007). We use an abstract object factory to manage the creation of instances of objects in the `GameObject` hierarchy. The factory keeps a reference to all instances it creates. This allowed us to extend the functionality of the factory and allow for searching and destroying of instances that were created by the factory.

3.4 Networking

Our therapy tools are typically single-user applications, with networking (such as in massively multiplayer online games) not an issue. However, during our initial research and planning of the project, 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 user's choices.

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. We hence implemented a simple peer-to-peer based network model with the intention to keep it as transparent as possible. 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 will take care of communication among peers. All relevant internal messages are forwarded to all connected peers. The philosophy is to make all hosts believe that all of the remote users are actually local.

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. Thus occurrences of synchronisation issues should be minimal. A possible 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. This would enable the engine to easily handle a classroom-like situation, with ten to twenty users.

3.5 Scripting

One of the objectives of this project is to provide a platform for the creation of educational games and ASD therapy tools without the need for extensive programming knowledge. A first step 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 (Ierusalimschy et al., 2006).

The development of educational games and therapy tools with scripting languages gives some advantages over traditional compiled languages (Ousterhout, 1998). Scripting languages are generally interpreted languages, which means we do not have the overhead of recompiling the program to change parameters or fix errors. An added benefit of interpreted scripting languages is that it allows us to design the engine to run a level or scenario as a script. Therefore, new content can be added to a game or tool, as scripts, without the need to recompile the engine.

In order to offer a versatile scripting component in the engine, we opted to use the simplified wrapper and interface generator (SWIG) (Beazley et al., 2009). SWIG allows one to link a C or C++ application to many different scripting languages by using interface files to define the API of the application. Using SWIG we have created Lua and Java wrappers for the engine.

Exposing the engine to scripting languages also paves the way for middleware that can be developed to simplify the development of games and therapy tools. The software on this middleware layer can be used to create a game or therapy tool, and the result would simply be scripts that would be executed by the engine. We are currently designing a visual programming environment on this middleware layer.

3.6 Graphical User Interface

We use a third party library for the creation, management and rendering of our graphical user interfaces (GUIs), namely, MyGUI (Evmenov, 2009). MyGUI is an add-on for OGRE and is being maintained by an active community of developers. The MyGUI library supports many common GUI widgets, such as windows, buttons and text boxes. MyGUI also includes a so-called layout editor: a point and click application for creating GUIs.

The appearance of MyGUI widgets is determined by so-called skins. These skins are customisable. This customisability is important, since individuals with ASD might find GUIs with certain traits (such as flashing icons) distracting and unusable (Lazar, 2007).

3.7 Replays and Data Capturing

Discussion with various therapists and teachers stressed the importance of feedback from the engine with regards to the actions of the child working with one of the therapy tools. We therefore implemented an extensive specialised action data capturing system in the engine. On a per-user basis, for each game or therapy tool, the engine provides a replay module. The therapists can use the replay facility to see the actions that a user took while playing a game or while engaged in a session with a therapy tool. We also support the data capturing of certain aspects of a game or tool, such as how many mistakes a user made before reaching the correct solution. Our data capturing system can easily define data capturing actions when creating a game or tool, so that every game or tool can have fully customised data capturing.

3.8 Multilingualism

Since ASD therapies are language and culture dependent, our engine must support a component through which games and therapy tools can be made available in multiple languages. To that end we include a language localisation component in our engine. This component operates in a similar fashion to GNU gettext (GNU, 2009). Files are used to map a phrase from the default language into a target language. During run-time the user can then select a desired language and the component will use the language files to display all text in the selected language.

3.9 Engine Implementation

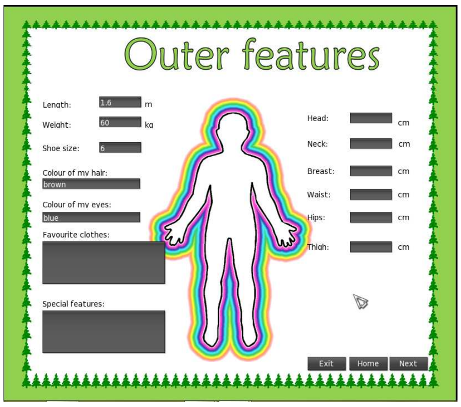
We implemented our engine as described in the design above. From the design it follows that adding new systems to the engine requires the creation of classes (that possibly extend some existing base classes in the engine) and making use of the message-passing model for communication. Following this approach to adding new features, the engine was implemented in various iterations, adding more functionality to the engine with each iteration. The inclusion of the physics system is an example. For the physics system the actor paradigm was introduced into the game object model, and a physics task was added to the engine. The success of this iterative approach provides an indication of the modularity and extendability of the engine.

After detailed modular testing, we continued with the implementation of games and therapy tools on the engine. The aim was to further test the implementation, and also to make sure that the design was appropriate. We briefly describe these test implementations in the next section.

4 GAME AND THERAPY TOOL IMPLEMENTATIONS

In order to test our platform we implemented two therapy tools and one educational game. The first therapy tool involved the computerisation of a number of worksheets from a popular workbook for individuals with ASD, namely *I am Special* (Vermeulen, 2000). This implementation was completely programmed in Lua and tested the scripting language interface, GUI and data capturing. A screen capture of one of the computerised worksheets is shown in figure 4.

We implemented a basic educational game, aimed at first grade learners, using our platform. The game was designed in partnership with a local school. The game presents learners with a number of exercises, similar to what they would encounter in a classroom situation. If the learners successfully complete an exercise, the story of the game advances. The logic of the game was implemented in Lua. All text in the game is available in multiple languages and in audio for those learners that cannot yet read at a satisfactory level. A further feature that is tested in the game is that of data capturing. All of the attempts and answers learners give to exercises are captured. In summary, the game tests the rendering capabilities of the engine, audio playback, input, language localisation and data capturing.



Length: 1.6 m
Weight: 60 kg
Shoe size: 6
Colour of my hair: brown
Colour of my eyes: blue
Favourite clothes:
Special features:

Figure 4: A screen capture of the *I am Special* tool.

A screen capture of the game is shown in figure 5. The game was tested at a local school for a period of 20 days during which no failures of the platform occurred. Teachers were very positive about its use and indicated that they found the data capturing aspect useful.

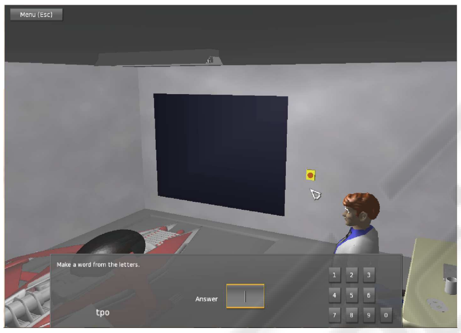


Figure 5: A screen capture from the Journey to the Moon game.

The second therapy tool we implemented is similar to virtual environments implemented by (Parsons et al., 2007) that aimed to teach social skills to individuals with ASD.

We implemented a restaurant scenario where the user must order food, pay for it and find a place to sit. A replay of the session is recorded. The tool also supports remote monitoring over the network, but at this stage no remote control or administration is possible. A screen capture of the restaurant scene is shown in figure 6.

5 FUTURE WORK

We are currently engaged in the design and implementation of a visual programming environment

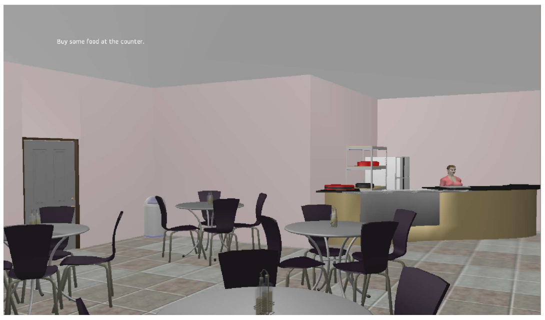


Figure 6: A screen capture of the restaurant scene.

which will be used to visually create games and tools based on the engine. The output of the program will be a set of Lua scripts that can be executed by the engine. Our visual programming language will be loosely based on other successful implementations such as Alice (Cooper et al., 2000) and Scratch (Maloney et al., 2004).

Another current project on the virtual environment platform involves the creation of an avatar with definable neurological characteristics, and a rule-based reasoning system to control the avatar's behaviour. We envisage the avatar being used in ASD behavioural research.

6 CONCLUSIONS

In this paper we presented a design and implementation of a generic and extensible 3D virtual environment engine that is intended to be used for the creation of educational games and therapy tools for individuals with ASDs. The engine is generic and extensible and fully supports various scripting languages, simplifying the development of games and tools and also increasing the general customisability of implementations created with the engine.

We implemented three different therapy tools with the engine, to serve as a basic test of the major functionality of the engine. In addition, we are in the process of developing a visual programming environment on top of the engine, to ease the development of further therapy tools.

All our results up to now indicate that our intended goal, namely, to develop a generic open-source 3D virtual environment platform, has been reached. It remains to develop more middleware, so that the platform can become more user-friendly for the development of additional therapy tools.

ACKNOWLEDGEMENTS

The financial assistance of the National Research Foundation under research grant number 65856 towards this research is hereby acknowledged. Opinions expressed in this paper and conclusions arrived at are those of the authors and not necessarily to be attributed to the National Research Foundation. The financial support of the Wilhelm Frank Foundation is also gratefully acknowledged.

REFERENCES

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

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

Cooper, S., Dann, W., and Pausch, R. (2000). Alice: a 3D Tool for Introductory Programming Concepts. In *Proceedings of the Fifth Annual Consortium for Computing Sciences in Colleges – Northeastern Region (CCSCNE), Ramapo, New Jersey, USA*. http://www.sju.edu/ scooper/alice/ccscne00.pdf.

Creative Labs (last accessed in July 2009). OpenAL. http://connect.creativelabs.com/openal/default.aspx.

Evmenov, G. (last accessed in July 2009). MyGUI. http://sourceforge.net/projects/my-gui/.

Fine, R. (last accessed in March 2008). The Enginuity Engine. http://www.gamedev.net/reference/programming/features/enginuity1/.

Gagne, G., Galvin, P., and Silberschatz, A. (2005). *Operating System Concepts*. Wiley.

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

GNU (last accessed in March 2009). Gettext. http://www.gnu.org/software/gettext/.

Goldsmith, T. and Leblanc, L. (2004). Use of Technology in Interventions for Children with Autism. *Journal of Early and Intensive Behavior Intervention*, 1(2):166–176.

Ierusalimschy, R., De Figueiredo, L., and Celes, W. (2006). *Lua 5.1 Reference Manual*. Lua.Org, Rio de Janeiro, RJ, Brazil.

Junker, G. (2006). *Pro OGRE 3D Programming*. Apress, Berkely, CA, USA.

Kerr, S. J., Neale, H. R., and Cobb, S. V. G. (2002). 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.

Lazar, J. (2007). *Universal Usability*. Wiley.

Linden Labs (last accessed in March 2009). Second Life. http://secondlife.com/.

Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., and Resnick, M. (2004). Scratch: a Sneak Preview. In *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing, Kyoto, Japan*, pages 104–109.

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

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

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

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

Streeting, S. et al. (last accessed in July 2009). Object Oriented Graphics Engine. http://www.ogre3d.org.

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

The Crystal Space Community (last accessed in March 2009). Crystal Space 3D. http://www.crystalspace3d.org.

The OGE Community (last accessed in April 2008). The Open Game Engine. http://www.opengameengine.org.

The Panda3D Development Team (last accessed in April 2008). Panda3D. http://www.panda3d.org.

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