USING GAME ENGINE TECHNOLOGY FOR VIRTUAL ENVIRONMENT TEAMWORK TRAINING

Stefan Marks[1], John Windsor[2], Burkhard Wünsche[1]
[1] *Department of Computer Science, Faculty of Science, The University of Auckland, New Zealand*
[2] *Department of Surgery, Faculty of Medicine and Health Sciences, The University of Auckland, New Zealand*
stefan.marks.ac@gmail.com, j.windsor@auckland.ac.nz, b.wuensche@auckland.ac.nz

Keywords:
Serious Game, Source Engine, Medical Teamwork Training, Head Tracking, Non-Verbal Communication, Head-Coupled Perspective

Abstract:
The use of virtual environments (VE) for teaching and training is increasing rapidly. A particular popular medium for implementing such applications are game engines. However, just changing game content is usually insufficient for creating effective training and teaching scenarios. In this paper, we discuss how the design of a VE can be changed to adapt it to new use cases. We explain how new interaction principles can be added to a game engine by presenting technologies for integrating a webcam for head tracking. This enables head-coupled perspective as an intuitive view control and head gestures that are mapped onto the user's avatar in the virtual environment. We also explain how the simulation can be connected to behavioural study software in order to simplify user study evaluation. Finally we list problems and solutions when utilising the free Source Engine Software Development Kit to design such a virtual environment. We evaluate our design, present a virtual surgery teamwork training scenario created with it, and summarize user study results demonstrating the usefulness of our extensions.

1 INTRODUCTION

In recent years, virtual environments (VEs) have become increasingly popular due to technological advances in graphics and user interfaces (Messinger et al., 2009). One of the many valuable uses of VEs is teamwork training. The members of a team can be located wherever it is most convenient for them (e.g., at home) and solve a simulated task in the VE collaboratively, without physically having to travel to a common simulation facility. Medical schools have realised this advantage and, for example, created numerous medical simulations within Second Life or similar VEs (Danforth et al., 2009).

To implement a VE, the developer has to choose between three possibilities:

- To use a completely implemented commercial or free VE solution like Second Life (Linden Research, Inc, 2010). This has the advantage of being able to completely focus on content creation instead of having to deal with technical implementation questions and problems. However, the disadvantage is that these frameworks cannot easily be extended with additional functionality required for a specific simulation scenario.
- To build a VE from scratch. This enables complete freedom in the design and usability of the VE, but significantly extends development time.
- To use a simulation framework that can be flexibly extended to account for special design requirements, but already provides a solid foundation of functionality to achieve a quick working prototype.

Whereas the first two options are located at the opposite extremes of the spectrum, The last option is located between these extremes in terms of development flexibility and rapid prototyping. A game engine, the underlying framework of computer games, can be used as such a framework. This is the principle behind "serious games":

To use the technology of computer games, e.g., graphics, sound, physical simulation, multi-user, but to replace and adapt the original content to build "serious" applications, e.g., for education, training, or simulation.

The literature provides several examples of studies with simulation environments based on game engines, e.g., Taekman et al. (2007), MacNamee et al. (2006), Smith and Trenholme (2009). For an extended review of serious games, see Susi et al. (2007).

However, rarely does the reader find information discussing design options, the advantages and disadvantages of tools such as game engines, and how to use them effectively and integrate new functionalities. This makes it difficult for researchers to extend existing simulations or create new ones. Exceptions are papers like Ritchie et al. (2006), where not only the used tools for the development process but also source code details are provided.

We have created a VE for medical teamwork training which provides additional control mechanisms by using a webcam to capture the head movement of the user. This head movement is decomposed into the translational part which is used for head-coupled perspective (HCP), and the rotational part which is used to control head gestures of the user's avatar to convey non-verbal communication cues. The results of our user studies show that HCP improves the usability of the VE as it introduces an intuitive view control metaphor that even inexperienced users were able to master within seconds. In addition, tracking-based head gesture control of the avatar improved the perceived realism of the simulation (Marks et al., 2011).

This paper provides insight into the design of game engines and their modification for advanced "serious games" applications. In particular we explain how new user interface devices can be integrated. In our discussions, we use the Source Engine (Valve Corporation, 2007) as example, which, at the time of the survey, fulfilled most of our simulation requirements: good graphical and animation capabilities, availability of an Software Development Kit (SDK) and developer tools, and reliable synchronisation of physically simulated objects among multiple clients. The details of the selection process of a suitable engine can be found in (Marks et al., 2007).

Section 2 presents the design of our VE framework. In Section 3, we describe the details of the implementation. A summary of the results of our user studies conducted with the framework are then presented and discussed in Section 4, and we finish with the conclusion in Section 5.

2 DESIGN

The goal of the research described in this paper is to utilise a game engine to implement a virtual environment for surgical teamwork training. An important component in teamwork is non-verbal communication, such as head gestures, which we capture with a webcam and then map onto avatars in the virtual environment. In addition we need an intuitive method for view control, since most surgical procedures require the surgeon to use both hands for instruments. We therefore decided to implement HCP using webcam input. HCP changes the view of the VE based on the movements of the user's head position in front of the monitor. The implementation of these additional hardware components and control metaphors is also part of this paper.

Realising our simulation scenario requires changing the game content, gameplay, and integration of webcam input to control game engine parameters. Figure 1 gives an overview of the architecture of the resulting system. The sections marked in red were developed, extended, or modified for the implementation. Figure 2 shows a screenshot of the final VE for teamwork training simulations.

Figure 2: Screenshot of the final surgical teamwork simulator *MedVE* created from the original deathmatch game code

The simulation is run on a central server that all users connect to with their client computers. The server as well as the clients run their part of the VE engine, being constructed on top of

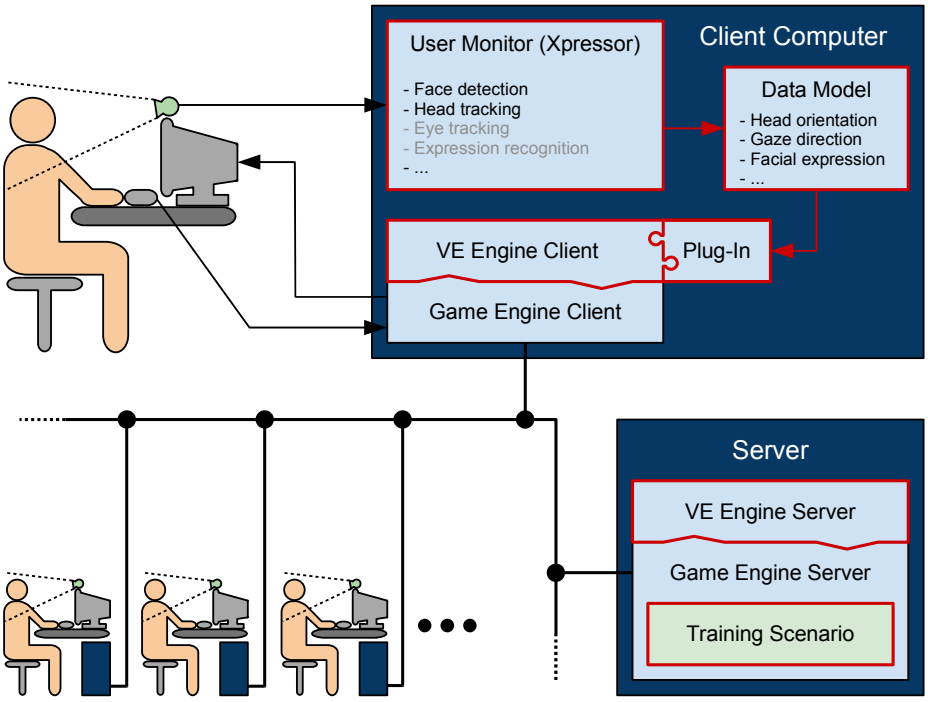

Figure 1: The functional blocks of the simulation framework

the Source Engine. It is important to distinguish between the terms "game engine" referring to components of the simulation framework that are parts on the Source Engine itself and therefore cannot be altered, and "VE engine" referring to components that are based on the Source SDK and have been altered to create a VE with new features and interactions.

The original game content on the server is replaced by the teamwork training scenario. This includes virtual rooms, objects, instruments, sounds, textures, 3D models, etc.

On each client, an additional program, called *Xpressor*, is running, using the input from the webcam for tracking the user's head and face. The tracking information is sent in the form of a specific data model (see Section 3.3) to a plug-in of the VE engine. By using an external tracking program and the plug-in architecture, it is easily possible to exchange these components later with more advanced ones, without having to modify the actual VE engine.

The translational head tracking information is used to control the view "into" the VE. This so called head-coupled perspective (HCP) enables intuitive control, such as peeking around corners by moving the head sideways, or zooming in by moving the head closer to the monitor.

The rotational head tracking information is used to control the head rotation of the user's avatar. That way, other users in the VE can see head movement that is identical to the movement actually performed physically by the user, such as nodding, shaking, or rolling of the head.

In addition, data from face tracking can be used to detect facial expressions and transfer them onto the user's avatar. Bartlett et al. (2008), for example, present a system that recognises a large set of movements of facial keypoints, such as lip corners, eyebrows, or blinking. Using a simpler set of movements and keypoints, Queiroz et al. (2010) created a virtual mirror, where an avatar mimics smile, gaze and head direction, and an opened/closed/smiling mouth of the user in realtime. In our implementation, we use a non-commercial version of the face tracking library *faceAPI* which does *not* include the detection of facial expressions.

3 IMPLEMENTATION

In the following three sections, we will explain the implementation of the three important components of this framework: the virtual environment, the user monitor *Xpressor*, and the data model.

3.1 Virtual Environment

3.1.1 Steam Client

The modification of a game that utilises the Source Engine starts off with *Steam*, a client software of the manufacturer Valve, designed to enable the user to buy games online, download and install them, and to keep the games updated when bugfixes or extras are released. Other features of the *Steam* client include:

- Digital rights management: It is not possible to install and/or operate games that have been acquired illegally.
- Social networking: A voice and text chat client enables users to communicate with friends to, e.g., organise an online game meeting.
- Data management: Configurations and saved games are not only stored on the local computer but also on the *Steam* servers. This enables users to log on to other computers and to still have access to their saved games and game configurations.

The disadvantage of having such a central system is the dependency on the provider. When *Half-Life 2* was released in December 2004, the huge demand on the servers brought the content delivery system down, resulting in massive criticism of Valve. It took the company some time to stabilise their service to a point where it could be considered user friendly and reliable again.

During the implementation phase, we had mixed experiences with *Steam*. Purchasing and installing the components that we used for the VE was unproblematic, easy, and fast. However, throughout the developing phase, updates were released that would repeatedly break not only our code, but also that of thousands of programmers on other projects worldwide. In those times, the forums were laden with emotional and even insulting posts. A solution to the problem would usually appear some time later, ranging from days to several weeks.

In one specific case, an update broke the map editor two weeks before we had planned to conduct a user study. Fortunately, at that time, all maps were already implemented, but the prospect of not being able to edit the maps in case of necessary changes was worrying. From that event on, as a precaution, we switched off the updating functionality when user studies were about to be conducted.

3.1.2 Creating a New Project

The Source SDK is available to anybody who has purchased at least one game that utilises the Source Engine, e.g., *Half-Life 2*, *Team Fortress 2*, *Portal*. With the help of the *Steam* client, the SDK is easily downloaded and installed like any other game.

The SDK offers a feature to create a new project, called modification. For the first experiments, we selected "Modify Half-Life 2 Multiplayer". The code that is installed afterwards forms a fully functional multiplayer game that can then be modified.

Using a game with similar functionality to the intended application, e.g., multi-player support, this gives a good starting point to experiment with the code and the game engine, and to start modifying certain aspects of the game. Several pages on the Valve Developer Community (VDC) website give additional hints and ideas, for example the "My First Mod" tutorial (Valve Developer Community, 2010c).

In theory, this process is straightforward and unproblematic. In practice, the SDK installer crashed at the end of the installation process. Several repeated attempts with different target directories did not prevent the installer from failing. However, after each crash, the code seemed to be completely installed. Therefore, we opened the project with Microsoft Visual Studio 2008 in order to compile the code. A first attempt ended with several error messages, caused by source files that were missing in the list of project files. This problem was easily solved by adding those files to the project. The second attempt caused a compiler crash that had to be fixed by changing some lines in the code, as described in Valve Developer Community (2010a).

Finally, after some additional but only minor problems, the code compiled and could be executed.

3.1.3 Version Control

Directly after creating and compiling the modification project, we put the code under version control, using Subversion (The Apache Software Foundation, 2011), as described in Valve Developer Community (2010f). That way, we were able to update the code with changes that were applied to the SDK later in the development process.

Figure 3 shows the version control tree towards the end of the development phase. At the

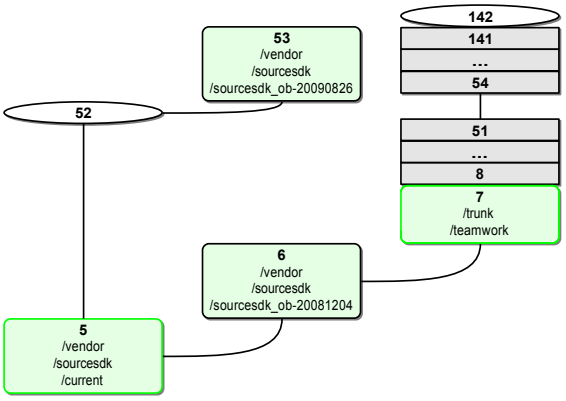

Figure 3: Version control structure of the project

beginning of the project (revision 5), we checked in the code as the original vendor code and tagged it with a timestamp (revision 6). From that tagged revision, we branched the working copy (revision 7) and worked on it for the duration of the project, checking in the changes after major alterations (revisions 8 to 51).

At the end of August 2008, Source released a major update of the SDK code. We checked in that code under an updated timestamp tag (revisions 52 and 53) and merged the *changes* from the old version (revision 6) to the new version into the main working directory. Afterwards, revision 54 contained the code we had changed since revision 7, *including* the changes from the old SDK to the new.

This merging process is the main advantage of maintaining a version control system: To be able to incorporate updates to the original code without having to worry about all the own changes to the project. Other advantages include the forced documentation of major changes and the ability to compare code of two revisions to get an idea of the history of changes to a file or a class.

3.1.4 Concepts of the Source Engine

Gaining an understanding of the design and functionality of the Source Engine was a very time-consuming process. At first sight, the developer website creates an impression of a thorough documentation. But when it comes down to details and specific questions, this documentation reveals large gaps and provides outdated or even contradictory information.

Throughout the development phase, our major source of information has been the Steam User's Forum, specifically the Source Coding Forum (Valve Corporation, 2010). Searches by keywords usually brought up helpful advice. An unspoken rule of the forum was to search thoroughly for an answer to your issue first, before you would post a question as a new thread. Not adhering to that rule would usually result in an insulting response, such as "[...] maybe if you bothered using the search bar you'd find this.", or in no response at all.

However, the majority of work necessary for understanding the Source Engine was to read through the provided code of the SDK, ignore several inconsistently implemented naming conventions, insert execution breakpoints, trace method calls through several class inheritance levels, and much more.

The quality of the SDK code is, in our opinion, not very high, considering the fact that this is a codebase that is delivered to developers worldwide. It gives the impression of being developed to a level that makes the final program work just fine and then not being maintained any more. We frequently found comments such as the following, confirming that impression:

- `Tony; the ugliest class definition ever, but it saves characters, or something. Should I be shot for this?`
- `FIXME: Do we want this?`
- `FIXME: Should this be local to c_env_spritegroup?`
- `FIXME: why is this called here? Nothing should have changed to make this [sic] nessesary`

Good documentation and a well structured codebase is important for any game engine that is to be the foundation of a simulation. Without these prerequisites, a lot of time is spent on deciphering the inner workings of the underlying code or on figuring out how to achieve a certain functionality, instead of implementing the essential parts of the program.

In the following sections, we will present some of the major concepts of the Source Engine that played an important role in the development and modification phase.

Game Engine/SDK Boundaries When a multi-user Source Engine game is started, four program parts are involved (see Figure 4):

- The game engine (`hl2.exe`) is executed, consisting of the server
- and the client part.

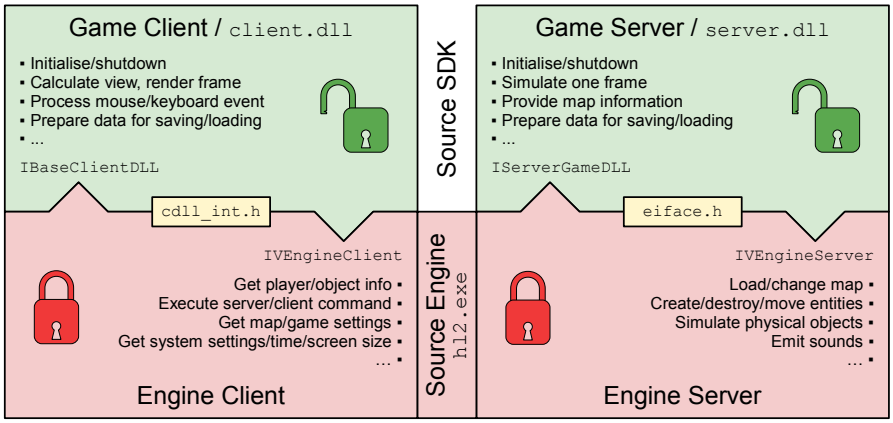

Figure 4: Boundaries between the Source SDK and the Source Engine

Depending on whether the user chooses to start a new game server or to connect to an existing game server, the engine then activates

- either the game server dynamic link library (DLL) `server.dll`
- or the game client DLL `client.dll`.

The game engine itself cannot be modified at all. No source code of the inner workings of the engine is given.

The SDK contains header files with interface definitions for the server (`eiface.h`) and the client part (`cdll_int.h`) of the engine. These interfaces provide access to very basic entity and resource management, and to the sound, graphics, and system functions of the engine.

It is possible to build a game completely from scratch, using only those header files. However, the SDK delivers a comprehensive set of classes and methods that, in its entirety, already constitutes a complete game. Starting from this point, the developer can now modify, remove or add custom parts to this framework. The advantage is rapid prototyping, as long as the result does not differ much from the original kind of game.

However, with every additional change that is necessary to get away from the original game towards the final product, it gets more and more difficult to implement the changes. Some of these difficulties are described in Section 3.1.5.

Client-Server Architecture Games and VEs for multiple users are mostly constructed using a client/server architecture (Valve Developer Community, 2010e). The basic principle of client and server communication of a game based on the Source Engine is shown in Figure 5.

The server is mainly responsible for running the simulation, updating the position, orientation, and speed of animated and physically simulated objects. In regular intervals, e.g., every 33 ms (=30 Hz), it receives compressed command packets from the clients, carrying information about mouse movement, keyboard input, and other events that the users on the clients have triggered. These command packets are unpacked, checked, and their effect is taken into consideration for the simulation: avatars move, objects are picked up or released, sounds are played, etc. After each simulation step, the new state of all objects and avatars is sent to the clients which can in turn update the changed state of the world on the screen.

During the runtime, each simulated object in the VE exists in two versions: One version, the "server entity", is managed on the server, and is actively simulated. The second version, the "client entity", exists on each client and is kept in sync with the server version by network variables (Valve Developer Community, 2010d).

These variables automatically take care of maintaining a synchronous state between the server and all clients. As soon as a variable value changes, its value is marked for transmission on the next update data packet from the server to the clients. To conserve bandwidth, the values are being compressed and only sent when they have changed. This mechanism is important to enable fluid gameplay on low-bandwidth connections, e.g., dial-up.

Prediction The fact that clients have to wait for a data packet from the server to show the updated world has a major drawback: Users would experience a noticeable delay to their actions, es-

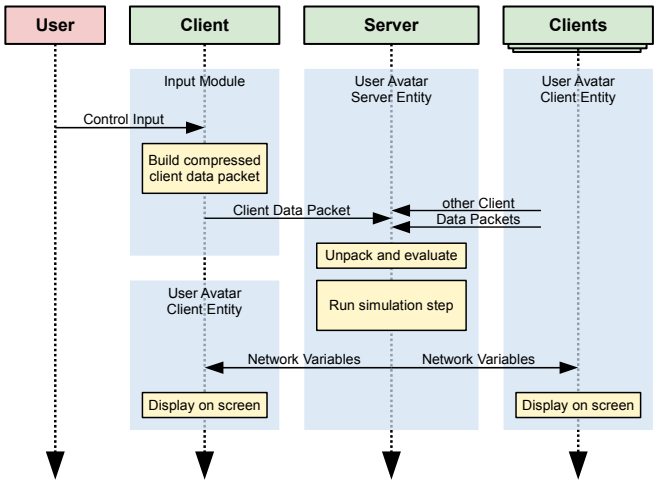

Figure 5: Data exchange between clients and the server of a multi-user game based on the Source Engine

pecially on slow network connections.

To avoid this delay and to provide a fast and responsive game, the client predicts the response of the server and uses this prediction for an immediate response to user input. When the client later receives the real server response, it corrects the prediction, if necessary.

For the prediction, the client needs to have the same rules and simulation routines as the server. In the SDK, this is implemented by a major duplication of code for the server and client entity representations. Instead of physical duplication, shared code is contained in shared source files (e.g., `physics_main_shared.cpp`) that are included in both, client and server projects.

Following the naming convention of the Source SDK, server classes are named `CClassName` and stored in a `server/classname.cpp/.h` file, while client classes are named `C_ClassName` and stored in a `client/c_classname.cpp/.h` file[1]. Shared code is usually stored in a `shared/classname_shared.cpp/.h` file.

Preprocessor directives are used to "rename" class names based on whether they belong to the server (`#define GAME_DLL`) or to the client representation (`#define CLIENT_DLL`), for example:

```
#ifdef CLIENT_DLL
  class C_TeamworkPlayer;
#else
  class CTeamworkPlayer;
#endif
```

or

```
// Hacky macros to allow shared code
// to work without even worse macro-izing
#if defined( CLIENT_DLL )
  #define CBaseEntity    C_BaseEntity
  #define CBaseAnimating C_BaseAnimating
  #define CBasePlayer    C_BasePlayer
#endif
```

Inarguably, this reduces the amount of typed code. But on the other hand, it increases the complexity, and reduces the maintainability and, most of all, the readability of the code. The preprocessor "renaming" made it sometimes impossible for us to trace through the inheritance structure of client classes, because it would always end up in the source code files for the server classes.

3.1.5 Stripping Down the Engine

The next big step, after understanding the engine, was to strip the project code of unnecessary classes and entities, e.g., weapons and the player health indicator. This step proved very difficult due to numerous interdependencies within the code. Weapon related code especially, was very deeply integrated into basic classes. Removal of one class file would break several other classes. It required a lot of re-compilation passes and uncommenting of code sections with `#if 0 ... #endif` preprocessor constructs, until the code would compile again.

3.1.6 SDK Inconsistencies

Several times, we stumbled over inconsistencies in the SDK code. As an example, the client representation of the player (`class C_BasePlayer`) has a dedicated method `bool ShouldDraw()` to

[1]The SDK code itself, however, does not follow that naming convention very strictly.

decide whether or not the player model should be drawn at all. The model has to be drawn when the user is in third person view, but not when the user is in a menu at the start of the game. When that method returns `true`, the `DrawModel()` method is called, and displays the player model on the screen.

During the development phase, we wanted to add "body awareness" (see Section 3.1.9), so that the user can see their own avatar's body when looking down. In theory, this required only a small change in the method, making it return `true` not only in third person view, but also in first person view. Interestingly, the addition did not result in any changes.

After some time of method call tracing, we discovered that the `DrawModel()` method also contained some conditional statements that prevented the model from being drawn. After we had moved those conditional statements into the semantically correct location inside of the `ShouldDraw` method, the player model was drawn as expected.

3.1.7 Changing the Interaction

One major change in the original SDK deathmatch game style was the primary interaction type. After we had removed all weapons, we wanted to assign the left mouse button click to grabbing and releasing of physical objects, and to triggering of interactions with objects, e.g., buttons or patients.

This seemingly simple change required a lot of reworking in the code to create access methods to the objects that the user interacts with, to enable users to take objects from each other, and to log all of those interaction events.

On a visual level, we wanted the avatars to grab an object with the right hand as soon as the user would pick it up. This can be implemented with inverse kinematics (IK): When the target position of the hand is given, IK calculates the position of the animating bones of the arm so that the attached hand reaches that position exactly.

The Source Engine is capable of IK, as can be seen in *Half-Life 2 – Episode 2*, where a certain tripod character always touches the ground with all three feet. However, Valve Developer Community (2010b) states that in multi-player games, IK is not activated due to difficulties and performance reasons on the server.

Our queries in the developer forums resulted in a confirmation that the engine is capable of IK, but nobody was able to give an answer on how to do it.

For this reason, grabbed objects "float" in front of the avatar while they are carried around. However, this flaw in realism did not distract the participants of the user studies. Some of them even made fun of the strange appearance, mentioning "Jedi-powers".

3.1.8 Changing the User Interface

Together with the change of the interaction style, we redesigned parts of the user interface. Among these changes was the replacement of the crosshair with a viewpoint indicator. In the original SDK, the centre of the screen is marked by a crosshair, indicating the point where the weapon would be fired at.

With the removal of any weapon related code, the crosshair turned into a viewpoint indicator. After some experiments with different indicator styles, We chose a segmented circle that turns green as soon as an interactive object is in focus, and closes when a physical object is grabbed and held (see Figure 6). Such a circle has an improved visibility over, e.g., a simple point. It is also less associated with weapons than, e.g., a crosshair.

The original weapon crosshair was simply painted at the centre of the screen. With the inclusion of head tracking however, we had to extend that part of code with a calculation of the point that the user would look at, considering the eye position offset caused by head rotation and translation.

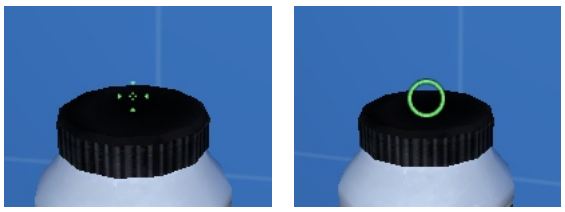

Figure 6: Different styles for the viewpoint indicator

3.1.9 Body Awareness

In the original SDK, the user cannot see the avatar's own body when looking down, as shown in the leftmost screenshot in Figure 7. To create body awareness, we had to change several aspects:

1. The body model has to be drawn when the game is in first-person view (see Section 3.1.6 for problems with the code concerning this aspect).

2. The camera viewpoint has to be synchronised with any animation of the body model, e.g., walking, standing idle. To achieve this, the camera position is constantly updated with the position of the eyeballs of the avatar model.
3. When looking up or down, the vertical head rotation cannot simply be translated into a camera rotation, because in that case the user would be able to see the inside of the head or the body (see left screenshot in Figure 7). We added a forwards translation to the camera that is slightly increased when the user looks up or down. Together with correct settings for the near and far plane of the camera frustum, this creates a realistic body awareness without literally having "insight" into the avatar model.

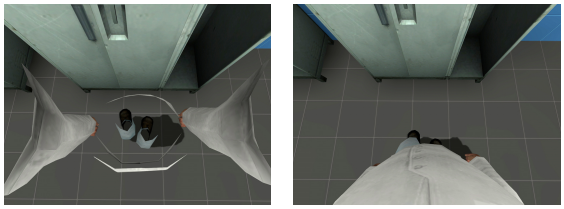

Figure 7: Creating body awareness for the avatar

We had planned to use IK to visualise the head movement caused by head tracking. Physical, translational head movement of the user would then have resulted in identical translational upper body and head movement of the avatar. As a result, an avatar would lean forward or sideways in sync with the user who is controlling it. However, we were not able to implement this feature due to the difficulties with IK described in Section 3.1.7.

3.1.10 Textures

The original textures of the SDK are designed for creating games that are set in a post-war era. These textures are, in general, worn down and dull, creating a depressive feeling in all maps created with them.

We replaced some of the wall, floor, and ceiling textures with synthetic textures that look like clean tiles. The regular style of the tile textures creates a very organised, sterile look. The realism of the rooms created with these textures could be increased further by using photos of real rooms. However, this was not a priority for our research, but it is an indicator of the complexity of creating realistic environments.

Figure 8: Examples of a room with the original Source SDK textures (left) and the custom textures for the user studies

3.1.11 Data Logging

We also implemented a data logging module that records user head movement, user interactions, and gaze targets and duration. The generated logfiles enable us to analyse individual and teamwork scenarios for statistical evaluations. An additional benefit, especially for teamwork assessment, is the ability of the logfiles to be imported into external assessment tools, like the behavioural analysis tool *Observer XT* shown in Figure 9 (Noldus Information Technology, 2010). This import eliminates the need for human assessors to observe a teamwork recording again to create a list of actions and behaviours. All this information is already present in the VE engine during the simulation and can therefore be directly exported into the logfile.

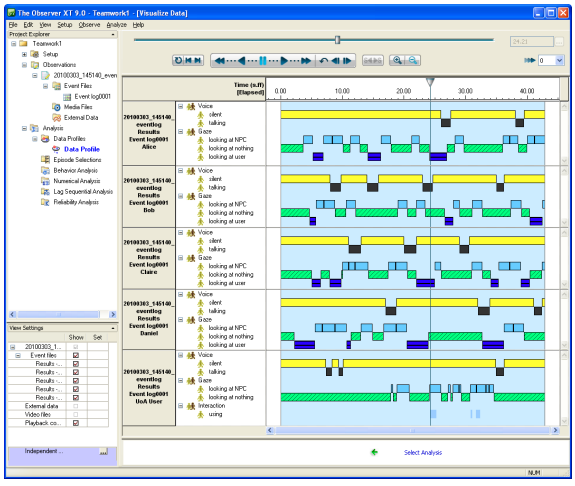

Figure 9: *Observer XT* visualising interactions, movements, and talk patterns of a teamwork simulation

3.2 Xpressor

Xpressor is the program that we developed for encapsulating the head tracking library *faceAPI*.

The program communicates bidirectionally with the VE engine, using two local user datagram protocol (UDP) connections.

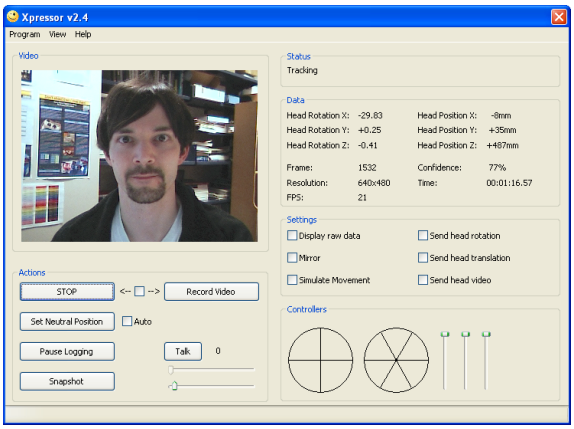

Figure 10: Screenshot of the user interface of *Xpressor*

The communication with the VE engine occurs through a plug-in, as shown in Figure 1. The Source SDK has certain settings and abstraction layers that prevent the direct use of networking functions and several other operating system related functions. However, it is possible to load plug-in DLLs and to exchange data with them. We therefore created a simple *Xpressor* plug-in that is loaded in the beginning, accepts the UDP connection, and relays the following data into the VE engine:

- translational and rotational tracking data,
- a low resolution video stream,
- information regarding whether the user is speaking or not, and
- values to control the facial expression of the avatar.

The video stream is helpful for the user e.g., to adjust his or her position at the beginning of a simulation. To conserve bandwidth, the video is resized to 100×60 pixel, converted to 4 bit greyscale, and transmitted with 10 fps via a separate UDP connection.

The program also monitors the signal strength of the connected microphone, signalising the VE engine via a flag whether the user is speaking or not. The state of this flag is determined by a simple signal energy threshold algorithm.

Xpressor is written in C++, using the Microsoft Foundation Classes (MFC) for the graphical user interface (GUI) (see Figure 10). For the control of the facial expression, we developed a custom circular controller interface, visualising six expression types as circle segments and the strength of the expression by the distance of the controller position from the centre of the circle.

While sitting in front of the screen, the user inadvertently shifts his or her neutral head position relative to the camera. As a result, any concept relying on an absolute position will reflect that drift in a slowly changing view of the VE. Similar to the recommendations from Sko and Gardner (2009) for games using HCP, we have implemented a configurable automatic slow adjustment of the neutral position towards the average of the measured position over several seconds. This adjustment accommodates for the gradual change of the neutral position and rotation of the user's head. To avoid an unwanted compensation when the user is at the extreme ends of the tracking range, e.g., when looking at an object from the side, the adjustment is reduced towards the outer regions of the tracking volume.

3.3 Data Model

The data model is a description of how to pack the values from head tracking and future facial expression recognition into a data structure that can be easily extended, but at the same time also easily compressed and transmitted.

Each parameter of the data model has

- a unique ID, e.g., `HeadPosX`, `HeadRotY`, `ExpressionType`, `ExpressionLevel`,
- a predetermined value range, e.g., from -1 to $+1$, and
- a predetermined precision in bits.

When sending a parameter, the value is converted into an integer with the stated amount of bits, and the ID is converted into a simple 16 bit hash value. Especially for a data model with a large number of parameters with long IDs, this conversion results in a significant reduction of the data volume that has to be sent over the network. However, it is possible that two different IDs can result in the same hash code. Therefore, we check for such hash collisions in our implementation.

On the receiver side, the value is uncompressed back into the original range and the ID hash is used to find the parameter to update with the new value. The precision has to be chosen so that rounding errors have no visible or accumulative effect.

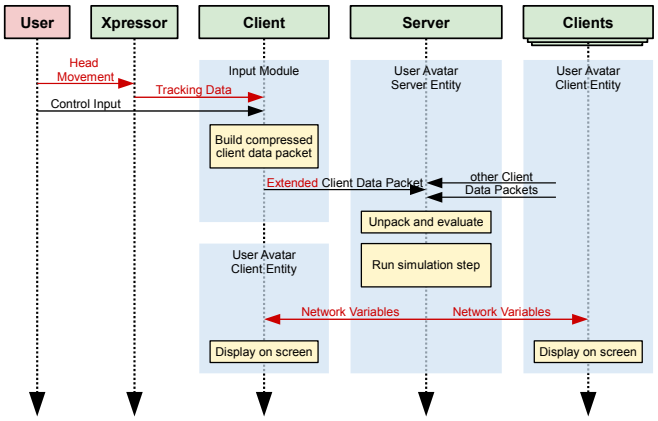

Figure 11: Data exchange between *Xpressor*, the VE clients, and the server

The data model is hard coded into *Xpressor* and the VE engine. It cannot be extended in runtime.

Figure 11 visualises the extension in the data flow between the VE clients and server. Because of the fast local UDP connection between *Xpressor* and the client, the data is transferred uncompressed, Between the clients and the server however, bandwidth can be limited, therefore the parameters are compressed according to the principle described above.

Different factors were influential for the choice of the value range and precision of parameters.

For the head rotations, we used the physical limitations of the human neck listed in Tilley (2002). Rotation around the X axis (nodding) is limited to $\pm 50°$, rotation around the Y axis (shaking) is limited to $\pm 60°$, and rotation around the Y axis (rolling) is limited to $\pm 54°$. A precision of 10 bit was sufficient to cause no visible steps in the rotation.

For the head translation, we chose a range ± 50 cm in all directions, which is sufficient for any head movement while sitting on a chair. A precision of 12 bit was sufficient to cause no visible steps in the translation.

Other parameters, such as facial expression type and level, usually were limited to a range between 0 and 1 and to a precision of 5 bit to 8 bit.

4 RESULTS

The modification of the Source Engine into a virtual environment for medical teamwork training with webcam support for HCP and head gestures was a non-trivial process due to the complexity and insufficient documentation of the engine, but allowed for rapid prototyping of early design stages.

All software outside of the VE engine, e.g., *Xpressor*, was kept modular, as well as most of the code we created to add functionality to the VE engine. This enabled us in the early stages of our experiments to easily exchange our own head tracking module by *faceAPI*.

However, features or modifications that required deep changes within the original code had to be kept close to the coding style of the SDK itself, resulting in suboptimal program code. The latter problem might be of a different magnitude when using different game engines, e.g., Unity 3D (Unity Technologies, 2011) that provide a more structured codebase to program against. The problems with the complexity of the code of the Source SDK were increased by insufficient documentation. A lot of development time was spent on deciphering the code or consulting the forums and developer websites for examples to compensate for the lack of documentation. To avoid this problem, it is important to put more emphasis on the quality of the documentation and the code of a game engine when engines are considered for selection.

Another time-consuming part of the development phase was the removal of unnecessary parts of the original deathmatch game (e.g., weapon code). Originally, this was the only codebase provided for the Source Engine. At about two years into the project, Valve provided a slightly modified SDK source code that allowed for the integration of additional aspects of the game: new player movements (e.g., prone, sprinting), player classes, teams, etc. However, this flexibility was implemented by adding a large amount

of `#ifdef ... #endif` preprocessor directives to the code, which was not improving the readability. Instead of providing this large configurable codebase, it would have been more helpful to have several different smaller examples instead, from which to choose the most appropriate one for the desired simulation. This is another factor to consider in the game engine selection phase: Is there a multitude of example programs from which to choose a suitable one as a starting point or that assist in understanding certain aspects of the engine?

Content for our VE was created using the free 3D editor *Blender* (Blender Foundation, 2011) and the tools provided by the Source Engine, e.g., the map editor *Hammer* and the character animation tool *Faceposer*. Most time during content creation was spent on figuring out ways how to simulate a specific effect or physical behaviour with the engine which is optimized for fast action gameplay, not for precise simulations. On several occasions, we had to compromise between realism and the ability of the engine to simulate a specific feature. One example is the bleeding that occurs during the surgical procedure we designed for the multi-user study. The Source Engine does not provide physically correct fluid simulation. Instead, we created a particle effect that resembles a little fountain.

We measured the "success" of the design and implementation of our VE indirectly by the user studies we conducted for our overall goal: to show improvements of usability, realism, and effectiveness of VE-based training scenarios by including camera-based non-verbal communication support and intuitive HCP-based view control.

Overall, the VE proved to be stable and intuitive to use for the participants, regardless if they were experienced in playing computer games or not. Our studies comparing manual view control against HCP showed that HCP is an intuitive and efficient way of controlling the view, especially for inexperienced users (Marks et al., 2010).

For highest user comfort, it is important that the delay between physical head movement and virtual camera movement is as short as possible. Our framework was able to deliver a relatively short response time of about 100 ms. However, this delay lead to participants repeatedly overshooting their view target. We suspect that the delay is a sum of several smaller delays in each processing stage of the data flow, therefore requiring several different optimisation steps for an improvement.

For our latest user study, we created a surgical teamwork training scenario and alternated between HCP and avatar control being enabled or disabled to investigate the effect of tracking-based avatar head movement on non-verbal communication within a VE. The results showed an increase in perceived realism of the communication within the environment (Marks et al., 2011). An effect on teamwork training effectiveness was not proven, but might have been masked by the experiment design. A clarification is subject to future research.

5 CONCLUSION

In summary, the Source Engine is suitable for rapidly developing a teamwork training VE, as long as the changes required to the original SDK code are not too major. The more functionality that is necessary for specific features of the desired VE, the more complex the coding task becomes. At a certain point, it would be infeasible to use this engine and alternative game engines would have to be considered.

However, the Source Engine proved stable and flexible enough for our medical teamwork training scenario with additional support for HCP and camera-controlled avatar head gestures. The user studies we have conducted show that these extensions are well received, and improve the usability and the perceived realism of the simulation. In addition, the digital recording of the interactions and behaviours within the VE is a valuable support for automated (e.g., with tools like *Observer XT*) as well as "manual" assessment of teamwork performance.

REFERENCES

Bartlett, M., Littlewort, G., Wu, T., and Movellan, J. (2008). Computer Expression Recognition Toolbox. In *Demo: 8th Int'l IEEE Conference on Automatic Face and Gesture Recognition*.

Blender Foundation (2011). Blender. `http://www.blender.org`.

Danforth, D., Procter, M., Heller, R., Chen, R., and Johnson, M. (2009). Development of Virtual Patient Simulations for Medical Education. *Journal of Virtual Worlds Research*, 2(2):3–11.

Linden Research, Inc (2010). Second Life. `http://secondlife.com`.

MacNamee, B., Rooney, P., Lindstrom, P., Ritchie, A., Boylan, F., and Burke, G. (2006). Serious Gordon: Using Serious Games To Teach Food Safety in the Kitchen. In *Proceedings of the 9th International Conference on Computer Games: AI, Animation, Mobile, Educational & Serious Games (CGAMES06)*.

Marks, S., Windsor, J., and Wünsche, B. (2007). Evaluation of Game Engines for Simulated Surgical Training. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 273–280, New York, NY, USA. ACM.

Marks, S., Windsor, J., and Wünsche, B. (2010). Evaluation of the Effectiveness of Head Tracking for View and Avatar Control in Virtual Environments. *25th International Conference Image and Vision Computing New Zealand (IVCNZ) 2010*.

Marks, S., Windsor, J., and Wünsche, B. (2011). Head Tracking Based Avatar Control for Virtual Environment Teamwork Training. In *Proceedings of GRAPP 2011*.

Messinger, P. R., Stroulia, E., Lyons, K., Bone, M., Niu, R. H., Smirnov, K., and Perelgut, S. (2009). Virtual Worlds – Past, Present, and Future: New Directions in Social Computing. *Decision Support Systems*, 47(3):204–228.

Noldus Information Technology (2010). Observer XT. `http://www.noldus.com/human-behavior-research/products/the-observer-xt`.

Queiroz, R. B., Cohen, M., and Musse, S. R. (2010). An extensible framework for interactive facial animation with facial expressions, lip synchronization and eye behavior. *Computers in Entertainment (CIE) - SPECIAL ISSUE: Games*, 7:58:1–58:20.

Ritchie, A., Lindstrom, P., and Duggan, B. (2006). Using the Source Engine for Serious Games. In *Proceedings of the 9th International Conference on Computer Games: AI, Animation, Mobile, Educational & Serious Games (CGAMES06)*.

Sko, T. and Gardner, H. J. (2009). Human-Computer Interaction — INTERACT 2009. In Gross, T., Gulliksen, J., Kotzé, P., Oestreicher, L., Palanque, P., Prates, R. O., and Winckler, M., editors, *Lecture Notes in Computer Science*, volume 5726/2009 of *Lecture Notes in Computer Science*, chapter Head Tracking in First-Person Games: Interaction Using a Web-Camera, pages 342–355. Springer Berlin / Heidelberg.

Smith, S. P. and Trenholme, D. (2009). Rapid prototyping a virtual fire drill environment using computer game technology. *Fire Safety Journal*, 44(4):559–569.

Susi, T., Johannesson, M., and Backlund, P. (2007). Serious Games -– An Overview. Technical report, School of Humanities and Informatics, University of Skövde, Sweden.

Taekman, J., Segall, N., Hobbs, E., and Wright, M. (2007). 3DiTeams — Healthcare Team Training in a Virtual Environment. *Anesthesiology*, 107(A2145):A2145.

The Apache Software Foundation (2011). Apache Subversion. `http://subversion.apache.org`.

Tilley, A. R. (2002). *The Measure of Man and Woman*. John Wiley & Sons, second edition.

Unity Technologies (2011). UNITY: Unity 3 Engine. `http://unity3d.com/unity/engine`.

Valve Corporation (2007). Source Engine. `http://source.valvesoftware.com`.

Valve Corporation (2010). Source Coding – Steam User's Forums. `http://forums.steampowered.com/forums/forumdisplay.php?f=195`.

Valve Developer Community (2010a). Compiling under VS2008. `http://developer.valvesoftware.com/wiki/Compiling_under_VS2008`.

Valve Developer Community (2010b). IK Chain. `http://developer.valvesoftware.com/wiki/$ikchain`.

Valve Developer Community (2010c). My First Mod. `http://developer.valvesoftware.com/wiki/First_Mod`.

Valve Developer Community (2010d). Networking Entities. `http://developer.valvesoftware.com/wiki/Networking_Entities`.

Valve Developer Community (2010e). Source Multiplayer Networking. `http://developer.valvesoftware.com/wiki/Net_graph`.

Valve Developer Community (2010f). Using Subversion for Source Control with the Source SDK. `http://developer.valvesoftware.com/wiki/Using_Subversion_for_Source_Control_with_the_Source_SDK`.