Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN EN BIO-INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN

Ondersteunen van het gedrag van smart entities op verschillende virtuele-omgevingsplatformen.

Thesis met als doel de graad te behalen van Master in de Toegepaste Informatica.

Dominique Dierickx

Promoter:	Prof. Dr. Olga De Troyer
Begeleider:	Dr. Frederic Kleinermann

Academiejaar 2008-2009

Vrije Universiteit Brussel

FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Supporting smart entity behavior across virtual environment platforms

Master thesis submitted in order to obtain a Master Degree in Applied Informatics.

Dominique Dierickx

Promoter:	Prof. Dr. Olga De Troyer
Advisor:	Dr. Frederic Kleinermann

Academic Year 2008-2009

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Frederic Kleinermann, for his personal guidance through the thought process that is behind this work, for sharing his knowledge in the field of 3D computer graphics and for his help with this document. I would like to thank my promoter, Prof. Dr. Olga de Troyer for her assistance during this entire academic year.

I would also like to take this opportunity to thank my parents for supporting me and my choices in every possible way.

Samenvatting

De dag van vandaag wordt er steeds meer gebruik gemaakt van interactieve driedimensionale omgevingen. Universiteiten bieden virtuele rondleidingen aan van hun campussen, winkels gebruiken virtuele etalages en sociale netwerkapplicaties laten gebruikers toe een virtuele hand te schudden met vrienden dichtbij of ver weg.

Het ontwerpen van zulke rijke virtuele omgevingen kan echter een zeer tijdrovende en repetitieve taak zijn. De informatie over de objecten in zulke omgevingen is vaak hard gecodeerd en dit zorgt ervoor dat hergebruik ervan op eenzelfde of een ander platform niet vanzelfsprekend is.

In deze thesis zal een methode besproken worden die het toevoegen van gegevens aan 3D modellen mogelijk maakt en die het uitvoeren van het gedrag van deze modellen kan realiseren op verschillende platformen.

Een virtueel object dat informatie bevat die nodig is om het object visueel voor te stellen alsook extra informatie over het object zelf wordt een Smart Entity genoemd. Deze extra informatie kan de gebruiker vertellen wat het object kan doen maar kan ook bijkomende gegevens over dat object meedelen aan de gebruiker. Deze gegevens zijn niet beperkt tot tekstfragmenten maar kunnen ook multimedia-bestanden zijn (afbeeldingen, audio, video, etc.).

Door deze informatie los te maken van de applicatie kunnen ontwerpers reeds bestaande entiteiten hergebruiken en kunnen deze op verschillende platformen en in verschillende werelden weergegeven worden, zonder dat deze gegevens verloren gaan.

Een applicatie zal worden voorgesteld die gebruikers toelaat zelf deze Smart Entities te creeëren en een prototype van een uitbreiding van een bestaande VE speler, die gebruik maakt van onze methode, zal worden besproken.

Kernwoorden: Virtuele omgevingen, smart entities, gedrag, animatie, interoperabiliteit, annotaties

Abstract

More and more, 3D virtual environments are used to deliver a rich user experience on the computer screen. Universities are offering virtual tours around the campus, shops are using virtual stores and social networking applications let you shake virtual hands with friends nearby or far away.

From a virtual world designer's point of view however, creation of content can be a repetitive and time-consuming task. Furthermore, the semantics of virtual entities are often hard-coded in the virtual environments, hindering reuse and interoperability.

In this thesis, a method is presented that allows adding semantics to a 3D entity and that allows invocation of the inherent behavior of these entities on different platforms.

Virtual entities that in addition to their geometric representation, also contain data on their semantics have been called smart entities. These semantics can state properties of the entity in question and can describe what the user can do with an entity and how these actions are to be performed. They can also contain more information on the entity, by providing links to related resources or even media content.

By decoupling these semantics from the virtual environment application, designers can reuse existing entities and port them to different platforms, while maintaining the semantics of the entity.

An authoring tool is presented that supports the creation of Smart Entities and a proof of concept VR player extension is discussed that takes advantage of the presented techniques.

Keywords: Virtual environments, smart entities, behavior, animation, interoperability, annotations

Contents

List of Figures

Glossary of terms

Behavior: The actions or reactions of an object or organism, in relation to the environment.

Digital Content Creation (DCC): The creation and modification of digital content, such as animation, audio, graphics, images or video, as part of the production process. To facilitate this process, a set of DCC tools are available.

Interoperability: A property referring to the ability of diverse systems and organizations to work together.

JavaScript: JavaScript is a scripting language used to enable programmatic access to objects within other applications. It is primarily used in the form of client-side JavaScript for the development of dynamic websites.

Mesh: A collection of vertices, edges and faces that define the shape of a polyhedral object in 3D computer graphics.

Platform: A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

Reflection: The process by which a computer program can observe and modify the structure and behavior of itself or other programs.

Rendering: The process of generating an image from a model using a computer.

Serialization: A term for converting a data object into another structure that can be stored on or transmitted over a medium. Deserialization interprets such data structures and transforms them into data objects.

Serious games: A term used to refer to an application developed with game technology and game design principles for a primary purpose other than pure entertainment. The serious adjective is generally appended to refer to products used by industries like defense, education [1], scientific exploration, health care, etc.

Smart Entity format: The Smart Entity file format is a platform-independent specification of the entity and its abilities. It contains the semantics that are attached to a 3D model as well as a reference to its visual representation.

SOAP (Simple Object Access Protocol): SOAP is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks.

XML (eXtensible Markup Language): The XML is a general-purpose specification for creating custom markup languages. It is classified as an extensible language, because it allows the user to define the mark-up elements. .

URI (Uniform Resource Identifier): A URI consists of a string of characters used to identify or name a resource on the Internet.

Virtual Environment Application: A software application which uses a Virtual Environment Player to display and control a VE and contains the logic of the VE.

Virtual Environment Player: A software application dedicated to the visualization of a VE.

Virtual Object (VO): A virtual representation of an object inside a virtual world.

Virtual Reality (VR): A field of study that aims to create a system that provides a synthetic experience to its user(s).

Virtual World (VW): A composition of virtual objects, the user(s) can navigate and interact with the world.

Webservice: A software system designed to support interoperable machine-to-machine interaction over a network.

WSDL (Web Service Description Language): The WSDL is an XML-based language that provides a model for describing webservices.

Chapter 1

Introduction

This chapter will start by introducing the main context of this thesis. Thereafter, both the motivation and intention of this work will be stated and the structure of the thesis will be discussed.

1.1 Virtual Reality and Game engines

Virtual reality (VR) is a set of technologies which allows users to interact with a computer-simulated environment. The user should feel immersed in this simulation, regardless whether the environment is completely imaginary or based upon the real world. Originally, the term VR referred only to immersive VR. In immersive VR, the user is fully contained in the virtual environment and all actions that the viewer performs are reflected in the virtual environment. Needless to say, these kinds of systems are technically very complex to create. Later on, the definition of VR has been widened to adopt both desktop VR and semi-immersive VR.

As opposed to what many people think, VR is used extensively nowadays for all kinds of specific applications. Typical examples are computer games and movies but several other fields have seen the benefits of using VR.

Surgeons use anatomic simulations to study or to prepare for complex operations [2], while safety instructors can design and test fire prevention and escape procedures using VR technology [3] [4]. VR has also been used in psychological research, because experiments can be repeated exactly and because each aspect of the testing environment can be fine-tuned. Especially the field of phobia treatment [5] has benefited from using VR technologies.

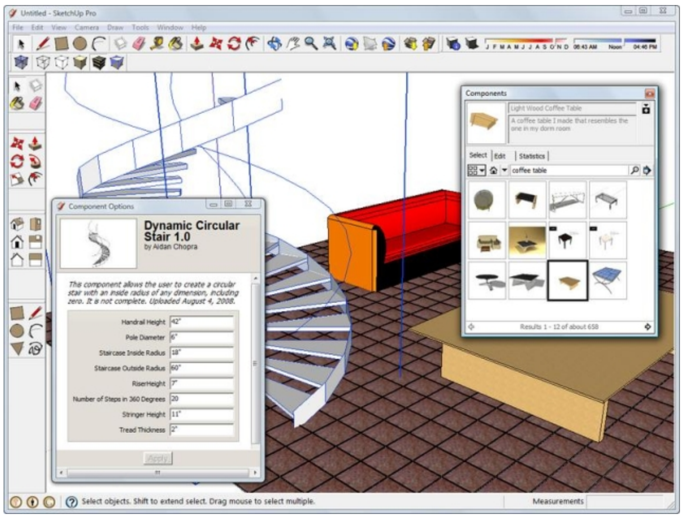
As computers are becoming more powerful, VR can be realized on normal desktop machines and even on mobile devices. During the last ten years, game engines have been created in order to ease in the development and deployment of games. Today, these game engines have most of the features that are needed to develop VR-applications, except support for high-end interaction and display techniques. Thanks to this evolution, the Virtual World (VW), including its objects and their behaviors, can be visualized and controlled by a game engine. This is the aim of the serious games initiatives. This is a term used to refer to a software application, developed using game technology and game design principles for a primary purpose other than pure entertainment. Examples of such purposes could be education, science, health care, etc. For this reason, in this thesis, the term Virtual Environment (VE) application will be used. This can be a VR-application which is running on the desktop or even in the webbrowser, but it can also be a game. Both of them can be developed by using (existing) game engines.

The VE is made of a virtual world (VW), which contains a number of virtual objects (VOs). These objects can have certain behaviors and can interact with each other and with the user(s). Users can also navigate inside the VW and can interact with the VOs by means of input devices such as the mouse and keyboard, joysticks or other devices.

1.2 Anatomy of a VE application

A VR application is composed of a number of key components. The following is an enumeration of these components, according to [6]:

1. **The scene and the objects**: The scene (or world) is the environment which hosts a set of VOs. These objects have their own boundaries, textures and physical properties. The scene also contains additional VOs like light sources and cameras, which are needed in order to visualize it. A scene also contains some specific behavior that it enforces upon its objects, i.e. the world's physics. The creation of VOs is typically done by using Digital Content Creation (DCC) tools, such as AutoDesk 3ds Max [7] and Google's Sketchup [8]. More information on these tools can be found in section 2.1.4.


Untitled - SketchUp Pro
File Edit View Camera Draw Tools Window Help
J F M A M J J A S O N D 06:43 AM Noon 04:46 PM
Component Options
Components
Select Edit Statistics
Apply
Measurements

Dynamic Circular Stair 1.0
by Aidan Chopra

This component allows the user to create a circular stair with an inside radius of any dimension, including zero. It is not complete. Uploaded August 4, 2008.

Handrail Height	42"
Pole Diameter	6"
Staircase Inside Radius	18"
Staircase Outside Radius	60"
RiserHeight	7"
Number of Steps in 360 Degrees	20
Stringer Height	11"
Tread Thickness	2"

Light Wood Coffee Table
A coffee table I made that resembles the one in my dorm room

coffee table

Results 1 - 12 of about 658

Select objects. Shift to extend select. Drag mouse to select multiple.

Figure 1.1: Creation of a living room using Google's Sketchup.

2. **Behavior**: Behaviors are attached to VOs, they describe what an object can do. The creation of realistic behaviors is one of the most difficult tasks in the design of VR applications. Most of the time, behaviors are implemented directly in the engine [9] by using a high-order programming language like C#, C++ or Java. Another approach is to use scripting languages [10] like Lua and Python [11] in order to separate the behavior definition from the engine. Other approaches have also been developed which allow a user to model behavior at a higher level [12].

3. **Interaction**: Without allowing interaction, the user would find the Virtual World (VW) uninteresting. Users need to be able to navigate through the scene, manipulate and use the scene's objects and receive feedback from their actions. At the same time, objects themselves need to interact with each other, for instance when two objects collide, the physics engine needs to perform this collision in a realistic way. It is also possible that the objects themselves respond to a collision by exhibiting some behavior (ex. shrinking).

4. **Sound**: The use of sound in virtual environments greatly enhances the credibility of the scene. It is the task of the sound engineer to provide realistic audio, given the state of the environment. To enhance the illusion of immersion, the sound itself can also vary according to the location of the user within the environment. For instance when an avatar talks in a basement, an echo could be produced. Surround sound techniques can be used to further enhance the credibility of the scene.

5. **Communication**: While the first VE applications were mainly offline, single-user systems, a lot of the contemporary solutions are being used concurrently by a large set of users. As an example, consider Second Life [13]. Tens of thousands of people are using the system at the same time and want to communicate with each other. The deployment of such a large collaborative environment involves tackling classical network issues such as latency and synchronization but also privacy and security problems.

An important aspect that influences the credibility of the scene is the way it is rendered to the display. In essence, the rendering process generates a picture, given the internal mathematical model of the scene. In the context of VEs, this process needs to be performed at a rate which appears to be fluid to the human eye (about 25Hz). Rendering is composed of a set of features. Each feature results in part of the resulting rendering image. One of the most important features is called shading. Shading attempts to depict depth and lighting in 3D models by varying levels of darkness. Since VE shading calculations need to be performed in real-time, complex physical calculations need to be avoided. This is why shading techniques often employ approximations and heuristics with little physical basis [14] in order to gain a performance advantage.

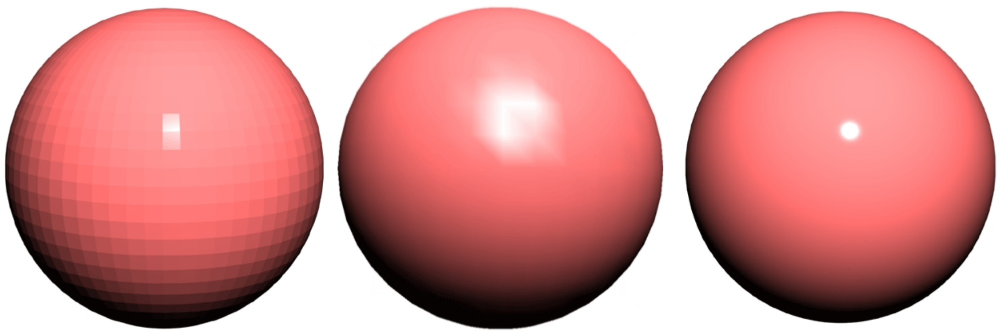


Figure 1.2: Common real-time shading algorithms: flat-, Gouraud- and Phong shading.

The creation of virtual environments and applications still remains a cumbersome and time-consuming task. This is mainly because it relies on so many different fields, including graphic design, sound authoring, artificial intelligence and of course software development. Luckily, the evolutions in both hard- and software have eased the creation of VE applications.

Before any scene can be visualized, the geometry of the particular scene needs to be modeled. A lot of different Digital Content Creation (DCC) tools are available that allow modeling rich 3D models (see 2.1.4), ranging from freeware applications such as Google's Sketchup [8], to high-end solutions like AutoDesk's Maya [15] and 3ds Max [7]. However, no DCC tool is the silver bullet. Each product has its own set of features, file format exporters and interfacing style [9]. The chosen platform will restrict the choice of compatible file formats and thereby also the set of appropriate tools. Luckily, most DCC tools provide a plugin mechanism that allows the addition of file formats. A large problem however is the quality of the different exporters. Especially third-party exporters often do not work correctly or do not export some features of the 3D model, such as embedded behaviors, annotations or parts of the model at all.

Many different platforms are available today (see 2.1) that take care of the visualization of 3D environments and expose an interface (API) that allows developers to control the virtual world. We will refer to these platforms as VE players. The choice of the VE player will impose some restrictions. For instance using Java3D [16] allows developers to target a large variety of different operating systems, while using Microsoft XNA [17] allows the creation of applications that run on the desktop, as well as the popular Microsoft's Xbox 360 console with a minimal amount of changes. Furthermore, several technologies are available which allow visualization of 3D environments inside webbrowsers. Examples of these are the eXtended 3D (X3D) players (like Vivaty player [18]) and Google's O3D framework (see 2.1). Therefore both designer and developers need to decide carefully which platform will be used, keeping into account which audience is to be targeted. This is not so easy, because they should also grasp all the constraints that are associated to the VE player.

1.3 Motivation

The motivation for the work presented in this thesis came from three main observations that were made at the beginning of the project and which mostly relate to annotations. Annotating is the process of adding extra-information to a Virtual Object (VO) and its behaviors. We will now explain these observations.

Observation 1: Annotations are only made for the static part of the VE and its VOs

Authoring tools are focusing mainly on the visual appearance of a VO and its behaviors. In recent years, these authoring tools have also been extended with the possibility to add extra information about a VO through the use of annotations, but this simple textual information is very basic and is only about the static part of a VO, so not on its behaviors. Furthermore, these authoring tools are storing the limited additional information in the same file which also contains the geometry of the VO (i.e. its visual appearance) and its behaviors. This is not a good thing, because the same VO could have different annotations attached to it, depending on the context it is in. For instance in the context of a video-game, a building could be annotated with text, stating that the enemies are inside this building. But in the context of urban design, this same building could be annotated with the text “IBM Office, no.9 Appleton Street, Boston”. To summarize: the appearance of a VO will remain constant, but the annotations can change depending on the context. Based on this, a number of research groups (see chapter 2) have been working on improving this annotation process by allowing designers to attach different types of content to a VO, such as text, images over even media content. Nevertheless, VOs are dynamic in nature, they can be interacted with and because they can have certain behaviors attached to them. For this reason, these annotations should also be about the behaviors and interactions. Most of the works on annotations have focused on the static structure of a VE and its VOs.

Observation 2: Most of the VE players do not exploit the annotations.

Although part of a VO has been annotated during the design, the VE players do not use these annotations, either to display to the end-users or to reason about these annotations in order to facilitate for instance the user's navigation inside the VE. At this moment, these annotations still need to be scripted or hard-coded. A good example is SecondLife (SL) [13] which shows videos, images, text about a VO. These annotations are contained with SL by means of a scripting language (the Linden Scripting Language [13]). This means that the designer needs to have some knowledge on programming. This may be one of the reasons why VEs still often lack in terms of annotations as it is still too difficult for designer to use them.

Observation 3: Portability of VOs and their related behaviors across different VE players is not well supported.

In the context of this work, the term portability will be used to indicate that a VO can be used on different VE players, while its visual representation and the behaviors that are attached to it remain the same. It is often the case that a VO and its behaviors work perfectly on some VE players, but not so well on others. As a result, the VO needs to be changed for different VE players. As the authoring tools nowadays support different geometry file formats, the static part of a VO can be generated in different formats. The problem of portability across multiple VE players can then be solved at least for the static part of the VO. But behaviors are often related to some VOs and they are often VE player dependent. As a result behaviors must be completely reimplemented into different languages in order for them to be supported by different VE players.

To overcome this problem of portability, it was thought that having a standard file format (like X3D [19] or COLLADA [20]) would overcome these problems. But this is still not the case as the developers of VE players do not always follow the specification of these file formats. As a consequence, behaviors often need to be reimplemented. For the designers of these VOs, this is really frustrating, because they often need to figure out ways of making their VOs portable. This can take a lot of time.

Based on these three observations, the main motivation behind this work is to explore a new research direction for supporting and using the annotations on a VO in a more efficient way so that they can be used to provide better feedback to end-users, resulting in higher user-satisfaction, but also to support portability across VE players.

1.4 Thesis aims and structure

1.4.1 Aims of this thesis

There are three main aims for this thesis:

1. Designing an approach which allows annotating the VOs of a VE with additional information. This could be information regarding the object in question or information about how the VO can be interacted with.

2. Developing a framework that uses these annotations so that VOs and their behaviors can be ported and used on VE players.

3. Based on our developed approach, an authoring tool will be developed with the aims of allowing non VR-specialists to annotate VOs and their behaviors and which will provide all the necessary information so that our framework can use it in order to support portability.

We will attempt to provide a methodology which will lift VOs from their purely geometric function and into a more semantically rich role. This lifting will promote the reuse and portability of these VOs on different VE players and in different VEs. VE players will have knowledge on the VOs which they use. This knowledge can be communicated to the users of the VE, who will be able to know more about the VOs they encounter in the VE. As a result, users will be more interested in the VE, because it will have more to offer them. Furthermore, we will explore a way in which VE players can exploit these additional semantics, so they will be able to perform the behaviors that are associated to a VO. Suppose we are modeling a car. VOs today only consist of their geometry, so we cannot annotate this car with the name of its manufacturer, its type, its reference manual, some real-life pictures, etc. There is also no way in which we can annotate a VO, representing a car, with the abilities it can perform, like opening the doors or trunk. Our approach will attempt to solve these two issues and will elaborate on a method which will allow the car and its associated abilities to be used on different VE players, without affecting the information that is available on the VO and the way the abilities of the car are executed by the VE player.

1.4.2 Thesis structure

Chapter 1 has introduced the context of this work as well as the motivation and aims of the thesis.
Chapter 2 will provide some background information related to virtual reality and 3D computer graphics and will explain related work in the scope of this work.
Chapter 3 will describe our approach: the “Smart Entity approach”.
Chapter 4 will present the proposed framework and our Authoring Tool.
Chapter 5 will provide a case study and will discuss the results and benefits of using our approach.
Chapter 6 will elaborate on future work
Chapter 7 will discuss the conclusions of this thesis.

Chapter 2

Related work

This chapter will start by providing some background regarding 3D computer graphics and VR in general. This section can be skipped by readers who are already familiar with VR. Later on, related work in the scope of this thesis will be reviewed.

2.1 Background

The development of computer graphics is a hard task which relies on a wide set of different fields, including graphic design, mathematics, physics, etc. In this section, some important concepts which have been used in this work will be briefly introduced. A more elaborate overview of computer graphics can be found in [14].

2.1.1 Virtual Environments

Techniques

Several techniques are available in order to visualize computed 3D simulations. A simple display can be used to show the simulation, this is called Desktop VR, but these displays cannot represent depth differences in a realistic and believable way. A number of companies, such as Philips [21] and Mitsubishi [22] are working on a flat display that can create a three dimensional illusion. Mitsubishi states that the next revolution of television is 3D TV. This surely seems to be a credible statement and the first results look positive, but it is not known when 3D TV will be available to the consumer's market.

Semi-immersive virtual reality takes things one step further. By using head-mounted displays, the illusion of a 3D environment is created. This is typically done using stereoscopic imaging. This technique consists of presenting a slightly different image of a scene to each of the user's eyes. When applied correctly, the illusion of immersion within a 3D environment will be perceived. The state of the art in the VR-world, however, is the

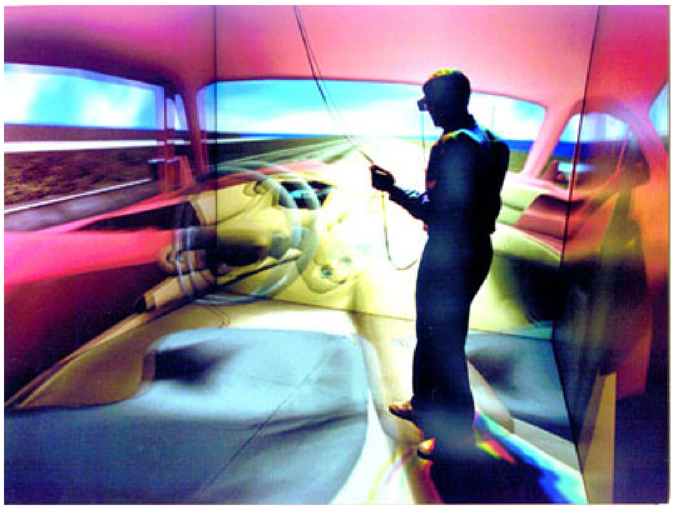


Figure 2.1: The CAVE system at the University of Michigan.

CAVE, or Cave Automatic Virtual Environment [23]. It is a fully immersive VR-system, developed at the University of Illinois, which surrounds the viewer with four projections: one floor- and three side-projections. A CAVE-user wears a set of stereo shutter glasses. These glasses alternately show images to each eye, synchronous with the refresh rate of the projection, thereby creating the 3D illusion. This is a stereoscopic imaging technique that is called alternate-frame sequencing. The movements of the user are monitored by using tracking systems and the simulation is updated according to these movements. Several universities around the world have a CAVE system that is used for various kinds of research topics. Some fields in which the CAVE system is employed include geology [24], medical & chemical research and biology. Next to its academic value, several projects have been created that use the immersive environment of the CAVE for entertainment purposes. At Brown University, the CAVE can be used by artists to make paintings and manipulate them using hand gestures [25] and of course 3D action games have also been ported to the CAVE system [26]. While the CAVE's research and entertainment value is obvious, the price of the system is still a large drawback.

Mixed reality

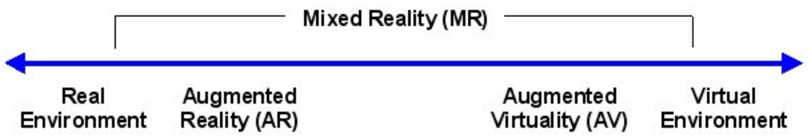


Figure 2.2: Paul Milgram and Fumio Kishin: The Virtuality Continuum

The merging of both real and virtual environments to produce a new environment in which physical and virtual objects can co-exist and interact is called mixed reality. There are two options, either the real world is extended with aspects of the virtual world, or vice-versa. These two approaches are respectively named augmented reality and augmented virtuality. An example of augmented reality are the head-up displays (HUD), which are being used in military aircraft but also in luxury automobiles. HUD's increase safety and efficiency by allowing users to maintain their original viewpoint on the real environment.

An interesting and practical example of augmented reality is the WikiTude AR Travel Guide [27]. This is an application for the T-Mobile G1, which runs on Google's Android operating system. It allows a user to target any landscape in the real world, using the built-in camera. The application will lookup any landmarks on the landscape using Wikipedia and overlay information on the visible landmarks on the display. This is feasible because the G1 phone has an embedded GPS chip and compass.

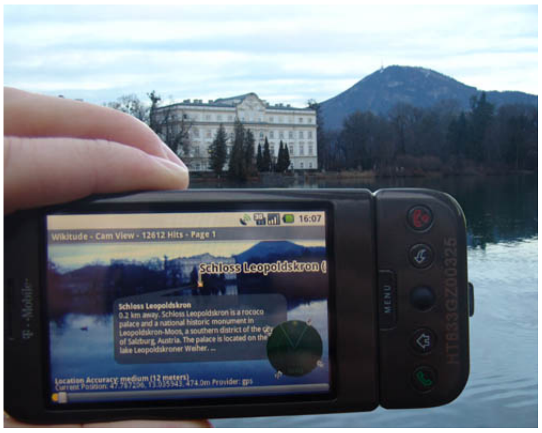


Figure 2.3: The WikiTude application in action.

2.1.2 3D Computer Graphics

The scene graph

A scene graph is an abstract data structure that represents the hierarchical and relational organization of the objects in a virtual world. The structure is typically represented as a graph or tree. The scene graph is composed of nodes, the nodes are objects that can represent either a 3D shape, a property or a node group. 3D Shape nodes are used to represent the actual visual objects and parts of objects in the virtual world. Property nodes can be used to describe objects.

The main purpose of using a scene graph representation is to ease object manipulation in the virtual world. For example consider a scene graph's fragment that represents a car. The car could have a node group called "wheels" that groups the four 3D shapes that represent the wheels of the car. By applying a transformation to the "wheels" node group, the transformation will also be applied recursively to the children of the group. Another advantage of the scene graph representation is that it allows addition and removal of nodes at runtime, so if any passengers would enter our car (like Frank and Rose), we can attach the passenger node group to the node group of the car and the passengers will move along with the car. Another interesting feature of the scene graph is called "node switching". Suppose we would like to change the 3D shape of the car's door when it has been hit. We can simply switch the 3D shape that represents the affected door, with a new 3D shape that is a visual representation of the damaged door. The scene graph is a very powerful concept, but the specific implementation can vary among different platforms.

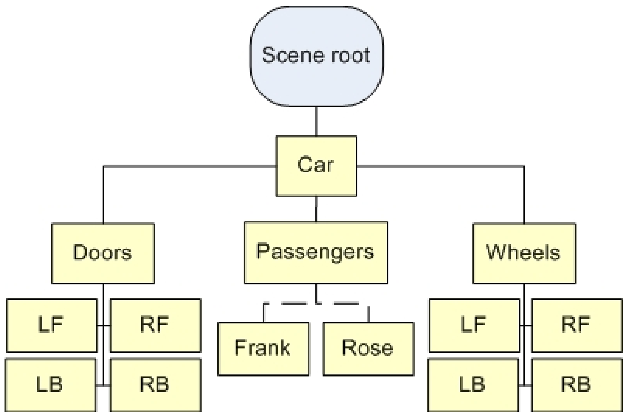


Figure 2.4: A scene graph describing a scene containing a car.

The camera system

A camera is essential to any virtual environments. Obviously, some sort of viewpoint to the world has to be specified before anything can be seen in that world. To define a simple camera in a 3D virtual world, we first need the following properties of the camera:

1. **Position**: A vector that defines where the camera is positioned in the world.
2. **Target**: A vector that defines which point the camera should be facing, the position and target vectors define the viewing direction of the camera.
3. **Up direction**: A normalized vector that defines which direction is up for the particular camera. Suppose we are looking at a chair in the real world, with only the position and target vectors known, we could still rotate our heads along the viewing direction, which results in many possible camera poses. In a typical virtual environment, the up direction would be $(0, 1, 0)$.

If we have these, our camera is said to be "uniquely defined". Before we can actually draw a virtual environment on the screen, we need to know how we should map the 3D system of our world to the 2D system of the display. This transformation process is called "projection". In order to draw

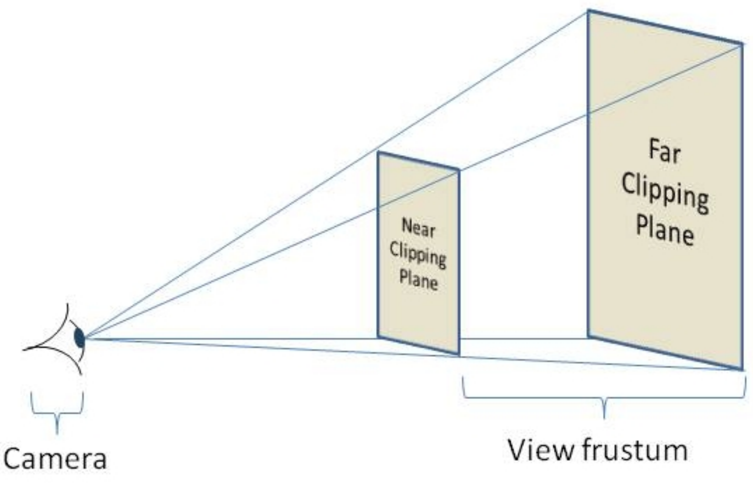


Figure 2.5: Depiction of the view frustrum.

the world, we define the view frustum (Latin for "piece cut off"). Anything that is within the frustum will be drawn by the engine, everything else will be clipped to increase performance (frustum culling). That is, anything further than the near clipping plane and closer than the far clipping plane. These planes are perpendicular to the camera's direction. The use of these clipping planes can result in reduced realism, because the viewer may notice that everything further than the far clipping plane disappears or is only displayed partially. The addition of fog can help soften this transition. The distance of the near and far clipping planes depends on the purpose of the VE.

Transformations

A transformation in 3D space ($\mathbb{R}^3$) is defined by a 4x4 matrix. The transformation of a single vector is performed by multiplying the matrix with the vector. Applying a transformation to a complex object such as a model, involves applying the transformation to each of its vertices. Transformation matrices are used to define the view and projection settings of the scene and to alter or animate the virtual world's entities. The main transformation matrices are the scaling, translation (moves a vector) and rotation matrices. This is the standard translation matrix in 3D space:

$$v' = T.v = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

The vector v is the original vector, the one we wish to translate. The parameters t_x, t_y, t_z are the coordinates we wish to add to the vector. Here is an example of a translation:

$$v' = T.v$$

$$v' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 7 \\ 3 \\ 1 \end{bmatrix}$$

In the example, the vector v has been moved 5 units along the Y (up) axis, resulting in v'. 3D platforms often include a set of available operations that allow the creation of translation, rotation and scaling matrices and the manipulation of general matrices. Several transformation matrices can be composed into one transformation matrix by multiplication. Matrix multiplication is performed right to left and not generally commutative, so one has to see to it that the used order is correct. Another thing to note is that one transformation may influence another transformation. For instance when two rotations around different axes are being combined, the axes of the second rotation will have been rotated by the first rotation. This is called “Gimbal lock”. A property of rotations is that two rotations can always be combined into one single rotation. Furthermore, any rotation can be represented by an axis (normalized 3D vector) and an angle of revolution around that axis. To store these two properties, a quaternion is typically used. A quaternion is an extension to the complex number system. The set of quaternions, $\mathbb{H}$, is equal to $\mathbb{R}^4$, a four dimensional vector space over the real numbers. The first three components represent the x-, y- and z-components of a vector in $\mathbb{R}^3$, these define the rotation axis. The fourth component (referred to as w), denotes the angle of the rotation. Three operations are defined on $\mathbb{H}$: Addition, scalar multiplication and quaternion multiplication.

$$\mathbb{H} = \{(x, y, z, w) | x, y, z, w \in \mathbb{R}\}$$

$$q = xi + yj + zk + w \quad (q \in \mathbb{H})$$

Since quaternions are so heavily used in the field of 3D graphics to represent a rotation around an arbitrary axis, several platforms expose a set of functions that allow users to handle quaternions without having to know the full mathematical extent of the topic. For example, both Java and XNA have classes ($Quat4d$, $Quaternion$) which have methods that perform quaternion operations and allow instantiation using the four components which were discussed above.

Coordinate system

In a virtual environment, all the contained objects have a position and an orientation in the 3D space, these are defined in a global 3D coordinate system. In the context of this thesis, the right-handed coordinate system will be used. In this system, the positive x-axis is pointing right, the positive y-axis is pointing up and the positive z-axis points towards the observer. Along with the global coordinate system, each object and part of an object can also have its own relative coordinate system. This coordinate system can be adjusted according to the purpose of the part. For instance when modeling a treasure chest, the designer could place the x-axis of the top of the chest on the hinges, so that whenever a rotation around the x-axis of the chest's top is performed, the chest would open up in a natural way.

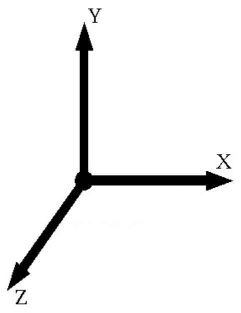


Figure 2.6: The right-handed coordinate system.

2.1.3 3D Model file formats

The creation and management of 3D virtual environments is overwhelmed by tools and file formats. Hundreds of different file formats are used, each with its ups and downs and dozens of high (and low) quality 3D authoring tools are available to content creators. In this section, some of the most relevant file formats that are available and used today are discussed.

X3D

The eXtensible 3D (X3D) standard was conceived by the Web3D Consortium [28] in 2001. It is an open standards file format which allows the creation of 3D scenes and objects by using an XML syntax. Since 2004, X3D became an ISO/IEC standard and it has become the leading technology for 3D in the webbrowser. X3D is the successor of the Virtual Reality Modeling Language (VRML) [29]. Along with the geometric structure of an object, it also provides mechanisms to store behaviors, interactions and meta-data, which are similar but more limited than the data that will be presented in this work. X3D represents a 3D scene using a scene graph (see 2.1) which contains primitive objects such as the Cube, Box and Cone objects and geometric transformations on those primitives. Some none-geometric objects, such as lights, cameras and even sounds can also be defined using the X3D syntax. The X3D format was created in order to deliver rich 3D content over the internet and into the webbrowser. To accomplish this, an X3D-viewer needs to be installed on the host system. Examples of such viewers are the Vivaty [18] and Octaga Player [30]. These players come with plugins for all popular webbrowser so that 3D content can be displayed within the browser. To create X3D files, users can use their favorite Digital Content Creation (DCC) tool to export to X3D if this option is available or use Vivaty Studio [18], which is a freeware authoring tool that was specifically designed for X3D development.

COLLADA

COLLADA is an attempt for establishing an interchange file format for interactive virtual environments. COLLADA is defined in an XML Scheme and was developed as a reaction to the various incompatible file formats that are around today. It was originally conceived by Sony Computer Entertainment in October 2004 to be the official format for the PlayStation 3 and PlayStation Portable game consoles. During its development, several other companies saw benefit in COLLADA and joined in, resulting in the Kronos Group consortium [20]. Members of this consortium are some of the biggest names in the (3D) entertainment sector, such as Blizzard, Dell, Sony, Barco, Creative, Electronic Arts, Google and many more. The COLLADA format was originally intended to be an intermediate format, to be used for transporting data from one digital content creation tool (DCC) to the other (ex.: 3D Studio Max to Blender) but some applications have used COLLADA as their native file format for storing 3D geometry. An example of this is Google Earth, which allows users to populate a map with a 3D object, using the COLLADA standard.

A standard such as COLLADA is a good thing for the digital entertainment industry, because it enhances interoperability between different platforms and because it can serve as the de facto standard for representing 3D objects. Support for COLLADA is available in almost any authoring tool, either native or using a plugin, but support in 3D API's is limited and mostly unofficial. Both Java3D and XNA have an unofficial COLLADA importer but these are far from production-ready and could not be used for the proof of concept. Hopefully COLLADA will evolve further and gain even wider acceptance in the future.

2.1.4 Digital Content Creation

Many different applications have been developed in order to ease the design of 3D models and animations. Each with its own ups and downs. Choosing the right tool for the trade is essential in order to increase productivity, especially for beginning 3D designers. In this section, some of the DCC tool's that have been used will be discussed briefly.

Sketchup

Google's Sketchup [8] is an award winning 3D modeling application that was developed by @Last Software. In March 2006, Google acquired this company and continued development of the application. The main advantage of using Sketchup is the low learning curve. Indeed, Sketchup boasts an amazingly simple, yet powerful click-and-drag user interface that has even been patented by Google. Best of all, Sketchup is completely free and available for download. There is also a professional edition which supports additional file formats and editing features. This version is required if the software is used for commercial purposes. Google Sketchup enables any user to publish 3D models to Google Earth or to a community repository called the Google Warehouse. This repository allows anyone to search for VOs, allowing even the layman to create rich virtual scenes. The application's professional version also allows exporting 3D models to a large set of different formats, including 3DS, OBJ, DAE (COLLADA) and FBX [31]. Sketchup does not supply any animation support by default.

Blender

Blender [32] is a 3D modeling application that has been released as free software under the GNU General Public License (GPL). Though it is costless, it offers a plethora of professional features to the user that are otherwise only available in high-end software. Blender is available for almost all popular platforms and boasts a large community of users. As oppose to Sketchup, Blender has a really steep learning curve. The use of keyboard shortcuts is almost imperative to working with Blender but once these keyboard shortcuts have been mastered, modeling with Blender can be much faster than with other commercial solutions. Blender offers export features to a lot of different formats, including 3DS, OBJ, DAE (COLLADA) and many more. Along with the creation of static models, Blender allows user to create rich (physics) animations, involving skeletons and inverse kinematics and even games.

3ds Max

For some time now, AutoDesk's 3Ds Max [7] (formerly known as 3D Studio Max) has been the de facto standard in the 3D modeling world. The application has a wide range of features, which seem overwhelming in the beginning but many game-content creators have sworn by it. Luckily a lot of free study material is available on the internet in order to help new users. AutoDesk features one of the widest sets of exporters that are available today, including 3DS, OBJ, DAE, X and FBX. Since the FBX format was created by AutoDesk [31], 3Ds Max's FBX exporter is the best available on any DCC tool. This makes the application an ideal DCC tool for the XNA platform. Its user interface is also a revelation as oppose to Blender, though some functionality is well hidden. It is also possible to create animations using 3ds Max, but its sister-product AutoDesk's Maya is especially created for that purpose.

2.1.5 VE players

Several VE players are available for developing and visualizing 3D virtual environments. In this section, a brief overview of the most used VE players will be given, along with some considerations with regards to ease of development, speed and presentation.

Java3D

Java3D [16] is an open-source, low-level 3D API that was created by Intel, Silicon Graphics, Apple and Sun Microsystems. It is not part of the Java Developer Kit (JDK). Java3D is an abstraction layer that can run on top of OpenGL or Direct3D and in the Java spirit, it is also platform-independent (though not for mobile devices). An interesting feature is the ability to create Java3D-applets. As all the other platforms, it supplies high-level methods for creating and manipulating the 3D geometry (using a scene graph representation). Any Java IDE can be used to develop Java3D applications, the only thing that needs to be done is adding the Java3D jar file to the classpath. Importers are available for different formats, including 3DS, OBJ, X3D and VRML [29]. Unfortunately DirectX, FBX or the industry standard COLLADA (DAE) file format support is unavailable. In January 2008, Sun announced that improvements to Java3D would be put on hold to focus efforts on integrating support with JavaFX to complement JavaFX's 2D scene graph. Nevertheless, Java3D is being used a lot today, though not that much information on it is available on the web (as compared to Microsoft XNA [17]).

Adobe Flash-based platforms

Since both the availability of broadband and the power of graphics cards have been increased during the last couple of years, the browser has become an ideal platform for the delivery of rich VR applications to the masses. Since Adobe Flash is one of the standards for creating Rich Internet Applications (99% of the web's users have Adobe's Flash player [33]), several third party attempts have been made to extend this platform with easy to use 3D capabilities:

- **Papervision3D [34]**: Papervision3D is the preferred engine for now because of its large userbase. Papervision3D delivers 3D virtual environments to every webbrowser with Flash support and has support for COLLADA files (including animations). Both of these features make Paperversion3D very attractive for web-based virtual environments.

- **Alternativa3D [35]**: A very impressive and performant Flash-based engine that is free for non-commercial use is Alternativa3D. It excels in the visualization of large, navigatable, environments and is already compatible with Adobe Flash 10. Alternativa3D can import 3DS and OBJ files, unfortunately COLLADA support is not yet available.

Since Adobe Flash 9 has no support for hardware acceleration, all of the graphics calculations of these engines are performed on the CPU. Flash 10 however will support hardware acceleration. Being able to use the full power the Graphics Processing Unit (GPU) will result in large speed benefits. One large drawback however remains that all assets (textures, models, sound, ...) still need to be downloaded from a webserver, resulting in longer loading times.

Google's O3D

In April 2009, Google introduced the O3D Application Programming Interface (API) [36]: a multi-platform solution that allows the visualization of 3D virtual environments in the webbrowser. Developing these environments is accomplished by using JavaScript. Programmers can manipulate the VW by modifying the scene graph. O3D provides a browser plugin for almost all browsers on Windows, Macintosh and Linux system. This plugin can either use Direct3D or OpenGL as its graphical layer, allowing it to perform all graphics calculations on the Graphics Processing Unit (GPU). O3D is capable of loading COLLADA models into the scene. These COLLADA models can be created using Sketchup Pro or any other DCC tool. Because this technology is still new at the time of writing, little documen-

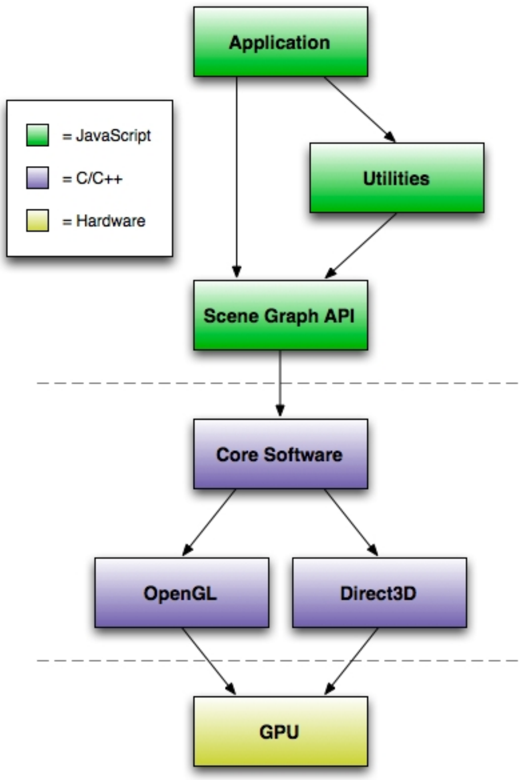


Figure 2.7: Google's O3D architectural overview.

tation is available. Developing VEs using JavaScript is also a hard task, since it offers limited debugging functionality. Nevertheless, this API has a great potential and cooperates perfectly with Google's Sketchup.

Microsoft XNA

Microsoft's XNA [17] (XNA is not an acronym) was released in December 2006. The main intent of the framework was to provide easy, yet powerful features to game developers. The framework is based on the Microsoft .NET framework and runs in a so called "managed execution environment". This means that the source code is first compiled to an intermediate code format (called managed code, comparable to Java's bytecode), this code is executed by a virtual machine. Execution environments are available for all popular Microsoft Windows versions and for the Xbox 360 gaming console. This allows developers to create games for the desktop computer and later on port them to the Xbox 360 console with little extra effort.

The XNA framework allows developers to have fine-grained control over all aspects of a game. HLSL (High Level Shading Language) effects can be used to program the graphics card's pixel, vertex and geometry shaders. Using HLSL, developers can add custom post-processing effects such as blurs, color effects, etc. but also complex lighting effects such as Gouraud and Phong shading. The power of XNA can be overwhelming to beginners. Fortunately there is a lot of documentation [37] available, there is community support [38] and a lot of great books [39] [40] have been published. The framework also integrates seamlessly into Microsoft Visual Studio, thereby enabling code completion, debugging and refactoring features. The 3D model formats that can be used are FBX [31] (AutoDesk) and X (Microsoft DirectX), unfortunately, only limited COLLADA support was available at the time of writing. Custom content format importers can be added by extending the Content Pipeline [41]. The Content Pipeline is responsible for reading data from the filesystem, parsing it and loading it into an XNA specific format such as the Model or Texture2D classes, so it can be used by the application.

2.2 Related research

Several papers have investigated means of adding semantics to virtual environments as a whole. In [42], the authors present means of creating Points of Interest (POIs) in an environment. These POIs can be associated with different kinds of content, including text, videos and images. This approach was developed in order to annotate the environments and the elements within this environment. The annotations are not bound to the objects but to the context of the object within a scene. The notion of Smart Objects has been used in [43], [44], [45] and [46] to denote objects which contain information on how they can be interacted with and manipulated. This information can be parts of the object which can be grasped and can define how an avatar should be animated while interacting with the object. In this work, some of the advantages of these added semantics are presented: by separating the animation specific information from the environment, the same model can be used in multiple applications. Furthermore, this notion of attached semantics promotes an object-oriented design, since each object encapsulates its own attributes and operations.

While in [43] and [44], the main purpose of the attached semantics is for object manipulation purposes, this thesis will focus on attaching semantics which describe the entity itself in more detail and which serve as a means of communication to the visitor of the Virtual Environment, hence Smart Entities. In [47], it is recognized that it is difficult to define a closed and sufficient set of behaviors, therefore our approach suggests a technique which allows the addition of new behaviors and even the modification of existing ones.

In [48], a framework is presented which focuses on the creation of rich behaviors by using both a graphical and a textual modeling language. These behaviors are instantiated by adjusting their parameters. This parameterization is also used in the presented approach. In this thesis, techniques will be presented which allow the attachment of parameterized behaviors to an entity and which allow the invocation of these behaviors on different VE players. In order to describe the semantics of our Smart Entities, the definition of Smart Objects in [47] has been used as a reference. The descriptions feature has been extended to include typing and visibility information and the focus is more on describing an entity's abilities and behavior rather than on actor-object interaction.

Chapter 3

The Smart Entity approach

This chapter will first review some important points which will explain the approach that was taken in this thesis. Thereafter, this approach will be elaborated further.

Note that in the context of this work, the term behavior refers to behaviors of levels zero, one and two in the behavior hierarchy which was proposed in [49]. Level zero relates to the direct modification of an entity's attributes such as its location, size and color. Level one behaviors define the modifications of level zero over time, resulting in animations. Level two behaviors are sequential calls to level one behaviors. The third and top-level behaviors are not in the scope of this thesis, as they involve high-level decision making techniques such as deducing what a VO should do at a particular time.

3.1 Overview of the approach

As discussed in chapter 1, there are three main aims of the presented approach. First, the approach should provide a way to express information about a VO and should allow VE players to take advantage of this additional information. Second, the approach should define a framework that facilitates portability of VOs and their behaviors by using this added information. Third, an Authoring Tool will be developed which conforms to our approach and enables designers to annotate VOs.

The Smart Entity approach is made up of two important elements. The first element is a file format which captures all kinds of information related to the VO through annotations. The need for this new file format originates from the observation that all DCC tools store the geometric definition and the behaviors related to VOs in a single file, but often in different formats (FBX [31], COLLADA [20], X3D [19], etc.). Furthermore, the formats that are used are often chosen in function of the VE player. Note that DCC tools which support annotations in a limited way, also store this information inside the file itself. They also store the behaviors that have been created by a designer in that same file. To summarize: the geometry of a VO and some limited semantics and behaviors are all bundled into one file. Another observation is that many VE players, which parse these files, were often developed before these file formats have been extended with the possibility to store these annotations. Therefore this information is almost never processed by the VE player. This is one of the reasons why VEs are still poor in terms of extra information other than information about the geometry of a VO, its visual appearance and its behaviors.

Storing these annotations within the same file as the visual representation and behaviors of a VO provides another disadvantage. The creation and maintenance of such VOs is not easy and needs to be done by experts that know their DCC tools well. Furthermore, the visual appearance of an object may not change often, but the annotations that are attached to it may change depending on the context the VO is in or even depending on the background of the end-user (culture, language, education, etc.).

Based on this, we have developed an approach which specifies a new file format which will facilitate the updates of this additional information and the portability of the VO and its information on different VE players. In other words, the VOs, which until now were only visual in nature, have been extended with relevant additional information. These VOs will be referred to as **entities**. The annotations will later on also be used to achieve portability of the VO and its associated behaviors across different VE players. The annotations will also be used to provide rich information about the VO to the user. By providing this information to users, they will be more attracted to the VE, since it has more information to offer them.

Users will also be able to know what the abilities of a VO are. By abilities, we mean that once a behavior is attached to a VO, the VO will have an ability to do something. For instance, suppose that a VO representing a door has two behaviors attached to it: opening and closing. We can then say that the VO has the ability to open and close the door. Furthermore, the behaviors can annotated so that the VE player will know how to invoke the abilities associated to that VO. This will also provide the advantage that VOs and VE users can know how to interact with each other.

For instance, the user represented by a human avatar, has clicked a door in a VE. The VE player can then lookup which abilities the door can perform and show a menu to the user, stating that the door can be opened or closed. This is the reason why we have extended the word entity with the adjective **smart**. As a result, VEs will become richer and they can become more useful and entertaining to their users because of the feedback users will receive. This may be a motivation towards developers of VE players to support this proposed format. Therefore, the second element in our approach is a framework that makes the assumption that existing VE players have been extended to support the proposed format and it will show how the promoted portability becomes feasible.

3.2 Smart Entity file format

As explained, our Smart Entity file format will contain information through the use of annotations about the VO and its behaviors. The information will be stated from both the users' point of view and from the VE player's perspective. Because our file format should be platform-independent, we have chosen an XML syntax to express its data. Most platforms have parsers that are more than capable of parsing our file format. The structure of the file format which has been formally captured by means of an XML Schema (XSD) file. This XSD file can be found in the appendix (8.1). To present an overview of the information that is contained into the format, the data model of the Smart Entity file format is displayed in figure 3.1. Each of the components in the data model will be described in more details now.

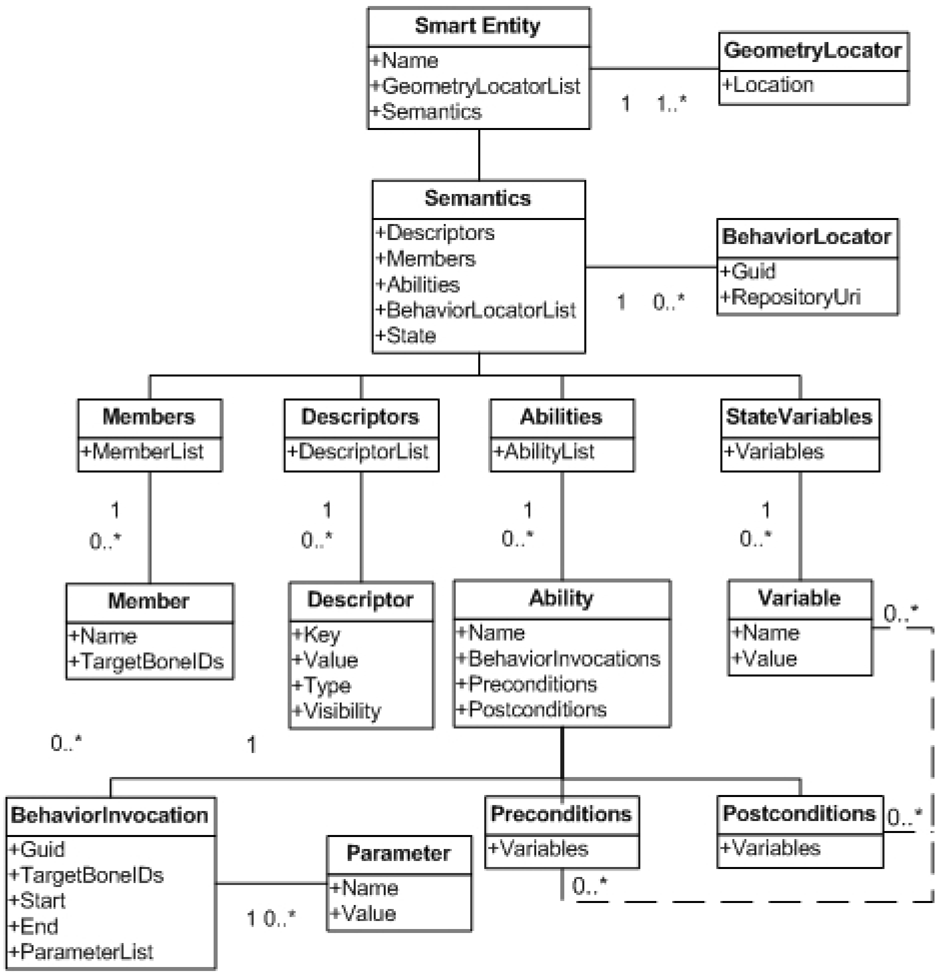


Figure 3.1: Data model of the Smart Entity file format.

3.2.1 Geometry Locators

Since the Smart Entity file relates to a 3D model, one of the first things that are required is a link to such a 3D model in some specified 3D model format. Because different VE players often also use different model formats, only linking to one model format may not suffice and may break down the interoperability we are looking for. For instance if platform A uses Autodesk's FBX format [31] as its format while platform B uses COLLADA [20] and our Smart Entity file only links to an FBX model, then the Smart Entity could only be used by platform A. This is why the presented format provides the possibility to link to multiple model files, by providing multiple URIs to the same 3D model, but in different formats. Designers can easily store the same model in different formats using the DCC tool of their choice, either by default or by using a plugin. This approach allows for a wider platform support of the created definitions. It is the task of the Engine Extension (see 3.3) to check which geometry locators are available and to pick a model format which is compatible with the specific platform. To summarize: our approach allows attaching the same 3D model in different formats so that the VE player can decide which 3D model it will load. These geometry locators are expressed in the XML file format based on the XSD schema (see 8.1) as follows:

```
<GeometryLocators>
 <GeometryLocator location="http://www.nokia.be/3d/nokia.X" />
 <GeometryLocator location="http://www.nokia.be/3d/nokia.X3D" />
 <GeometryLocator location="http://www.nokia.be/3d/nokia.DAE" />
</GeometryLocators>
```

Note that this example shows only a part of the Smart Entity file format. The VE player which parses the Smart Entity file format can choose which 3D model it will load, depending on the format of the 3D model. In the XML snippet, three URIs are available that point to a location on the network. This means that a VE player needs to download a compatible model first. Using URIs, we can also link to files on the local file system by using the "file://" prefix.

3.2.2 Descriptors

A good way of describing properties of a VO would be by using a key/value mechanism. For each VO, we can store a set of key/values that describe its properties from the VE users' perspective. Some examples are: (name, "Dominique"), (age, "23"), (gender, "male"). A key/value pair that describes a VO will be referred to as a descriptor. These descriptors are mainly intended to be shown to users of the VE but could also be used by the VE player itself. As an example of the use of these descriptors, consider the following scenario: an avatar, controlled by a user, walks around in a virtual museum. The user can click any painting in the environment and a menu will pop up that gives a brief summary of the selected work. The information shown is originated from the descriptors in a Smart Entity file, which links to the 3D model of the painting. The main usage of descriptors is to serve as a means of communication between the VO and a user, but they could also be used by the VE player internally. This indicates that there is a grouping that is needed. Some descriptors are to be exposed to users while others are to be consulted only by the VE player. This requires us to attach a visibility to a descriptor. Two types of visibility are needed:

- **Public**: A public descriptor is intended to serve as a communication to the users of the VE. They state relevant properties of the entity, such as a painting's artist(s) and style.

- **Private**: Private descriptors are to be used by the VE application. Examples are the entity's meta-data such as the author and the used DCC tool. Other examples could be points-of-interest (POIs) of the entity, like the location of a door's hinge.

From the examples that were stated above, we can deduce that simple text-based key/values are not enough. The values of properties should be typed, because the information they contain can be anything. It could be primitive content such as text, numbers or vector, but it can also contain links to resources (URIs), such as images, audio, video or documents. In order to give the VE application an idea of what content it can expect and to make the content machine-processable, some typing information is needed for each descriptor. Furthermore, it is necessary to agree on a set of common data types that are to be used.

Below is a list of the proposed data types.

- **Text**: Simple textual content such as the descriptors which were described in the beginning of this section.
  Example: (name = Dominique)
- **Boolean**: Represents a value which is either true or false.
  Example: (male = True)
- **Number**: Represents any real number.
  Example: (age = 23)
- **Vector2**: Represents a tuple of numbers.
  Example: (size = (22.4 , 12.3))
- **Vector3**: Represents a triple of numbers.
  Example: (hinge_location = (2, 4, 0))
- **Resource**: Represents a URI which links to a resource of an undefined type.
- **Image**: Represents a URI which links to an image.
- **Audio**: Represents a URI which links to a video file.
- **Video**: Represents a URI which links to an audio file.

The fact that the VE application is aware of these types allows different ways of communicating them to a user. For instance, a virtual showroom can contain various Smart car Entities. A user can select a car, upon which the descriptors of the car can be presented on the display. The user can view pictures of the car in various poses, watch a promotional video, open the car's reference manual in the webbrowser, etc. Even more, a comparison between a set of cars could be made by displaying all public boolean and number types of these cars in a table. An important aspect is that multiple Smart Entity files, which relate to the same 3D model, can coexist. This opens up possibilities for customization. Different Smart Entity files could link to the same 3D models. They could contain different descriptors. In our virtual showroom, the average visitor may not care about all technicalities of the car, such as its horse-power or the number of cylinders. While an enthusiast would want to know all this information. Another example could be a virtual museum. Different Smart Entity files could be created for each of the visitors' native languages. The VE player can then load the appropriate Smart Entity file, with regard to the visitor's language. As a result, information on the paintings can be tailored to the user's background and language. To conclude this section, an example of the descriptors will be given in the syntax of our Smart Entity file.

```
<Descriptors>
 <Descriptor visibility="Public" name="Title"
  value="Girl with a pearl earring" contenttype="Text" />
 <Descriptor visibility="Public" name="Artist"
  value="Johannes Vermeer" contenttype="Text" />
 <Descriptor visibility="Public" name="Period"
  value="1665" contenttype="Text" />
 <Descriptor visibility="Public" name="Location"
  value="Mauritshuis, The Hague" contenttype="Text" />
 <Descriptor visibility="Private" name="Dimensions"
  value="(44.5, 39)" contenttype="Vector2" />
</Descriptors>
```

The above XML example shows how descriptors are stored in our Smart Entity file format. These descriptors relate to a 3D model of a painting. It states public descriptors which contain information about the painting, which can be communicated to the visitor of the virtual museum. It also contains a private descriptor, which tells VE application how to position the painting in the virtual museum.

3.2.3 State

A VO will always be in a particular state, which is inherent to the VO itself. This state is the result of the sequence of abilities that have been performed on it. A door, for instance will initially be in the "closed" state, but when the ability which opens the door has been performed, the door will be in the "opened" state. To introduce this notion of state in our approach, a mechanism similar to the attributes in object-oriented programming has been employed. In the Smart Entity format, a number of state variables and their initial value will be defined. For our door, there would be only one state variable ("doorstate"), which will initially be set to "closed". These state variables can be asserted and changed by invoking abilities, as will be described in the next section. The following XML example shows how we can define the state of the door.

```
<StateVariables>
 <Variable name="doorstate" value="closed" />
</StateVariables>
```

This part of the Smart Entity file defines one state variable for our VO, namely the "doorstate". Its initial value has been set to "closed".

3.2.4 Abilities

One of the things that makes VEs fun to use is the interaction they offer between the user and the VOs. Users want to know what a VO is able to do and they want to invoke those abilities. In this section, we will introduce how these abilities are expressed in our Smart Entity format and how they can be used. Since we want to present the set of available abilities of a VO to the user, one of the first things that are needed to represent an ability is its name. This name should indicate to the user what will be done when the ability is invoked. In the VE, the user could select a VO in some way, for instance by clicking it, upon which the set of available abilities of that VO can be displayed. Users can then select which of the abilities of the VO they want to invoke. In this thesis, we will focus on explicit invocation of abilities by a user. Abilities could also be invoked automatically. For instance, when a collision between two VOs takes place, one VO could react by changing color. This work has not focused on this type of interaction, but suggestions on this topic, given the current approach, have been made in the Future Work chapter (chapter 6). When the notion of state was introduced in the previous section, the example of a door was given. Now we will elaborate further on this example. The set of available abilities of our virtual door depends on its state, i.e.: opened or closed. To express this in our format, we have used the idea of pre- and postconditions. Preconditions define what the values of the state variables should be if the ability can be invoked. For instance, the “open” ability of the door can only be invoked if the door is closed, i.e. when the “doorstate” variable of the VO has the value “closed”. To update the state of a VO after an ability was invoked, the postconditions can be used. Postconditions set the value of state variables after the ability has been invoked. When the “open” ability has been performed, the postcondition of the ability should set the state of the door to open. This is done by setting the value of the “doorstate” variable to “open”. The XML snippet below describes how these pre- and postconditions can be defined in the Smart Entity file format.

```
<Ability name="Open">
 <Preconditions>
  <!-- Compare the state of the VO -->
<Variable name="doorstate" value="closed" />
 </Preconditions>
 <Postconditions>
<!-- Set the state of the VO -->
<Variable name="doorstate" value="open" />
 </Postconditions>
 ...
</Ability>
```

When the VE player needs to present all available abilities to the user, it will enumerate each ability and check if its preconditions have been met. It does this by comparing the value of each variable from the state of the VO (as discussed in the previous section), with the value of that same variable which has been asserted in the preconditions section. If all of these comparisons succeed, then the ability can be invoked by the user. When an ability has finished, for instance when the door has been opened, the postconditions will be set. To accomplish this, the VE player's Engine Extension will enumerate all postconditions and set the state variables of the VO to the value that has been defined in these postconditions. So suppose we have invoked the "Open" ability, then the "doorstate" variable's value will be set to "open". As a result, the precondition for "open" will now fail and the "Open" ability will not be able to start again, unless the door has been closed and the "doorstate" has been set to "closed" by the close ability's postconditions. The usage of abilities in the Smart Entity format will be explained further in the following section.

3.2.5 Behaviors

Let us take a closer look at the abilities of a VO. Suppose we are animating a car. The car will have the ability to drive forward. To accomplish this, the wheels of the car will need to rotate and at the same time the car itself will need to be moved (translated) forward. We can say that this ability, "drive", is composed of five behaviors: the rotations of the four wheels and the translation of the car as a whole. Before this ability can be invoked, we need some information on how and when these behaviors will be performed. In the case of our car, the rotation and translation will start and stop at the same time. To spin the wheels, we need to rotate them a given amount of radians around their respective X-axes. To translate the car, we need to translate the entire car a number of units along its forward vector. We have now abstracted the behaviors away from their implementation. We have simply stated what should be done and which parameters are needed in order to perform the behavior. This abstraction allows us to define abilities and their set of behavior invocations in an abstract way, without having to know how these behaviors are performed by the VE player.

From the example that was given, we can deduce that each behavior has some required parameters:

1. **Start**: When the ability is executed, at what time will the behavior be invoked.
2. **Duration**: What is the duration of the behavior invocation.
3. **TargetBoneIDs**: On which bones (geometric parts) of the VO will the behavior be invoked. (ex. front wheel)

In addition to these three required attributes, a behavior invocation might need other parameters that define how it should be performed. Examples of these are the rotation angle of the wheels of a car and the direction and distance of the translation of the car. These parameters depend solely on the behavior and its requested invocation and are unrelated to any implementation of these behaviors. By only stating the parameters and their values, the concrete implementation of these behaviors has been abstracted away and we can express the abilities of a VO in a platform-independent way, this method will be referred to as the parameterization of a behavior. As an example, consider a linear rotation behavior. The required parameters are the start and duration times, the target parts of the model (TargetBoneIDs) and the axis and angle of the rotation. Given these parameters, the rotation behavior could be invoked regardless of the specific implementation. The behavior could be implemented in Java for the Java3D [16] VE player, C# for Microsoft's XNA VE player [17], JavaScript for Google's O3D [36] VE player or by using any other programming language. Given this fact, we can identify a behavior implementation by what it should do. We attach this identification to all VE player specific implementations of that behavior. This identification has been called a behavior Global Unique Identifier or GUID. For example, we might have some code in a Java Archive (JAR) file that contains specific code for Java3D and which performs a translation, given the correct parameters. We could also have JavaScript code which performs the translation in the same way, but on Google's O3D VE player. These two behavior implementations will then have the same GUID. This GUID is an important aspect in our approach and is essential in providing behavior portability and reusability.

In the Smart Entity file, the abilities of the VOs will be described. Within this ability definition, all of the behaviors which make up the ability will be instantiated. To illustrate how an ability is defined in our XML Smart Entity file format, an example is given below.

```
<Ability name="Drive forward">
 <Preconditions />
 <Postconditions />
 <BehaviorInvocations>
  <BehaviorInvocation guid="TranslationBehavior">
   <Parameters>
    <Parameter name="targetboneIDs" value="car" />
    <Parameter name="start" value="0" />
    <Parameter name="duration" value="10000" />
    <Parameter name="direction" value="0 0 1" />
    <Parameter name="distance" value="10000" />
   </Parameters>
  </BehaviorInvocation>
  <BehaviorInvocation guid="RotationBehavior">
   <Parameters>
    <Parameter name="targetboneIDs" value="lfwheel;rfwheel;
      lbwheel;rbwheel" />
    <Parameter name="start" value="0" />
    <Parameter name="duration" value="10000" />
    <Parameter name="angle" value="60" />
    <Parameter name="axis" value="(1,0,0)" />
   </Parameters>
  </BehaviorInvocation>
 </BehaviorInvocations>
</Ability>
```

In this XML example, we have define a car's “Drive forward” ability. As we have explained, this ability can be dissected into a translation of the entire car and rotations of the four wheels. In the snippet, we have created invocations of these behaviors. The translation behavior is referred to by the “TranslationBehavior” GUID and the rotation behavior by the “RotationBehavior” GUID. Now that we have stated which unique behavior we want to invoke, we can state the parameters for these behaviors. These parameters are bound to the GUID of the behavior. The Behavior Repository, which will be elaborated in the next chapter, allows designers to query for these parameters. For now, we will assume that these are provided as such. The three first parameters are required, while the other ones are specific to the GUID of the behavior.

3.2.6 Behavior Locators

In the previous section, we have described how we have abstracted away from the concrete behavior implementation by means of parameterization of the behavior invocations. Furthermore, we have used Global Unique Identifiers (GUIDs) to group behavior implementations which perform the same behavior on different platforms. To provide portability, the concept of Behavior Locators has been introduced in our Smart Entity file format. Here is an example of these Behavior Locators.

```
<BehaviorLocators>
 <BehaviorLocator guid="rotate"
   repositoryUri="http://localhost/BehaviorRepository.asmx"/>
 <BehaviorLocator guid="translate"
   repositoryUri="http://localhost/BehaviorRepository.asmx" />
</BehaviorLocators>
```

Behavior Locators link GUIDs of behaviors to a Behavior Repository. This Behavior Repository can be seen as a database which contains all VE player specific implementations of behaviors, grouped by their GUID. More information on this Behavior Repository can be found in the next chapter. These locators are used by our proposed extension of the VE player: the Engine Extension. This extension will attempt to find the specific implementation of the behavior for the VE player. This process is called Behavior Resolution. If a VE player needs to perform a certain behavior, the extension will query these Behavior Locators to find the Behavior Repository in which it can find the implementation. The extension will then ask the Behavior Repository for an implementation of the behavior for the VE player in question. This process will be elaborated in more detail in the next chapter. For now, it is important to know that the Smart Entity file indirectly contains the location of the VE player specific implementation of all behaviors that are used in the file.

3.2.7 Members

A VO can be composed of many different parts, we call these complex VOs [50]. To group a set of logically related geometric parts (bones) of a VO, a member can be defined in our Smart Entity format. A member is a named composition of strings which refer to the names of the bones that form the group. This feature was conceived in order to ease reuse of logically related bones. An example could be a “wheels” member that consists out of all of a vehicle’s wheels. Designers that want to attach a behavior to the wheels can refer to all four wheels by using the name of the member as the TargetBoneIDs parameter of the behavior, the Engine Extension will then replace this member by the bones it represents. Members cannot have the same name as a bone of the model. To illustrate how members are defined in our Smart Entity file according to the XSD scheme (see 8.1), an example is given below:

```
<Members>
 <Member name="wheels">
   <MemberBone id="wheelLF" /><MemberBone id="wheelRF" />
   <MemberBone id="wheelLB" /><MemberBone id="wheelRB" />
 </Member>
</Members>
```

3.3 Framework

In this section, we will describe the framework that uses the Smart Entity file format which has been discussed in the previous section. This format allows VEs and their VOs to provide information to the user and allows the user to interact with the VO, regardless of the VE player. In order to achieve this portability, VE players will need to be extended slightly. This extension will be elaborated on in this section. The framework which will be discussed here, consists of two parts: the Engine Extension and the Behavior Repository. These two components will also be discussed in this section. A conceptual overview of the entire approach is given in figure 3.2.

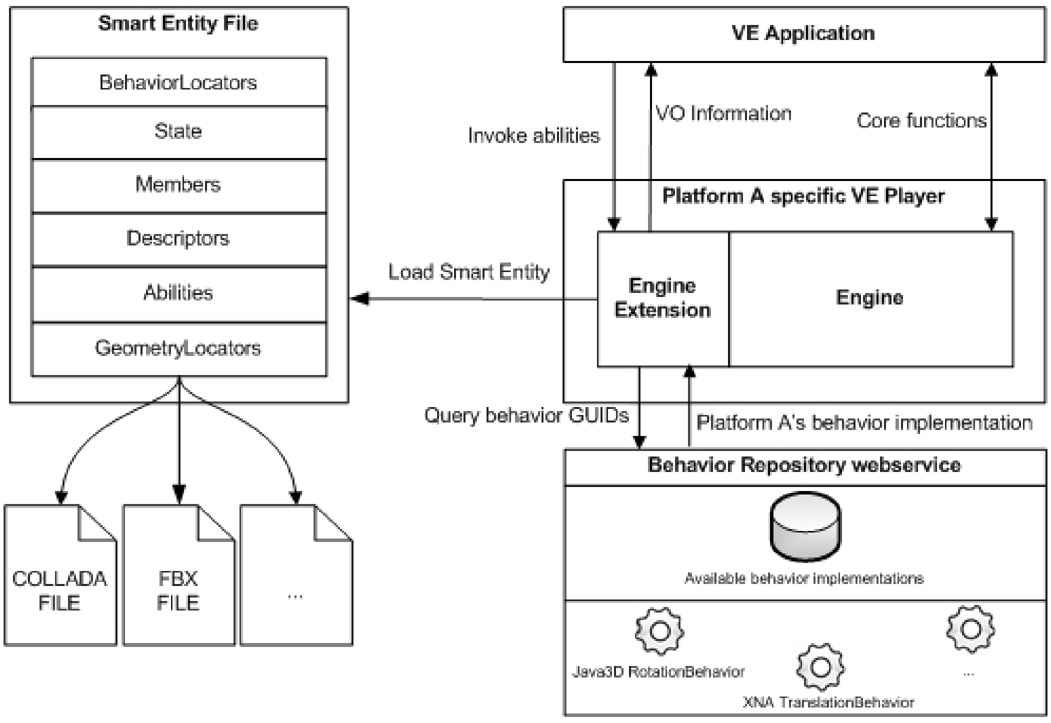


Figure 3.2: Conceptual overview of the architecture.

The main components of the framework are (1) an extension to existing VE players and (2) the Behavior Repository webservice. Each of these two components will be described in more details in this section. To design our framework, we have made two hypotheses. The first hypothesis is that developers of existing VE players will be willing to extend their VE players. We believe that we can persuade them to do so if they see all of the benefits that our approach can offer them and if the required changes are kept as minimal as possible. The second hypothesis is that the complete VE — its VOs and behaviors — will be annotated using our approach and that there will be a Smart Entity file for each VO in the VE. The next chapter will describe an Authoring Tool that has been developed in order to allow designers to annotate these VOs in an efficient way.

In the previous section, the Smart Entity file format, which is depicted on the left side of the scheme, was discussed. This section will focus on the right side of this scheme. An extension to the existing VE player will be elaborated. This extension will be responsible for loading the presented Smart Entity file format. The VE application can then display the information about the VO in the virtual world.

Furthermore, the VE application can ask the Engine Extension to invoke the abilities that were described in the Smart Entity file. In order to accomplish this, the Behavior Repository webservice will be queried (Behavior Resolution) for obtaining a behavior implementation that is compatible with the VE player and its platform. We will now discuss the components that make up the framework i.e.: the Engine Extension and the Behavior Repository.

3.3.1 The Engine Extension

In our approach, we have taken the view that existing VE players should be extended. This extension is necessary if designers want to use the benefits of the approach to their advantage: portability and reusability of the behaviors and information of VOs. We are not asking VE player developers to redesign their entire application. We only suggest a small extension to existing VE players. We know that the number of changes that are needed in order to adapt existing VE players for this approach, should be kept as low as possible if we actually want developers to implement it.

Let us begin by stating the tasks and requirements of our proposed Engine Extension:

1. **Loading the Smart Entity file**: The VE application can ask the Engine Extension to load a Smart Entity file. This file should then be parsed and transformed to an internal structure which can be used by the VE application.

2. **Behavior Resolution**: The Engine Extension needs to be able to find and load a compatible behavior implementation. In order to lookup a behavior that was designed to be used by the platform and VE player in question, the Engine Extension may need to contact the Behavior Repository webservice. This webservice will give the Engine Extension a behavior implementation that is compatible with the VE player and its platform.

3. **Reflection and invocation**: The Engine Extension does not know a-priori (at compile-time) which behavior implementations it will need, therefore it should be possible to load and execute external code at runtime (i.e. when the Smart Entity file has been parsed). The Engine Extension should also be able to pass the parameters of the behavior invocation from the Smart Entity file to the behavior implementation. The availability of these reflection routines is of the utmost importance for this approach to work. Fortunately, most programming languages (Java, C#, C++, ActionScript and even JavaScript [51]) have more than adequate routines which allow loading external code at runtime.

Behavior portability

It is time to elaborate further on how the approach achieves portability of behaviors. When a Smart Entity file has been loaded upon a request from the VE application, the Engine Extension will enumerate through all of the behaviors that are used in the abilities. For each unique GUID that has been found, the Engine Extension will check the Behavior Locators (see 3.2). These Behavior Locators should indicate the address of the Behavior Repository webservice in which the behavior implementation can be found. The extension can first check if it already has a local copy of the required behavior implementation, for instance by checking if a file with the same name as the GUID exists in a local folder (a local repository). If this is not the case, the Engine Extension should query the Behavior Repository webservice.

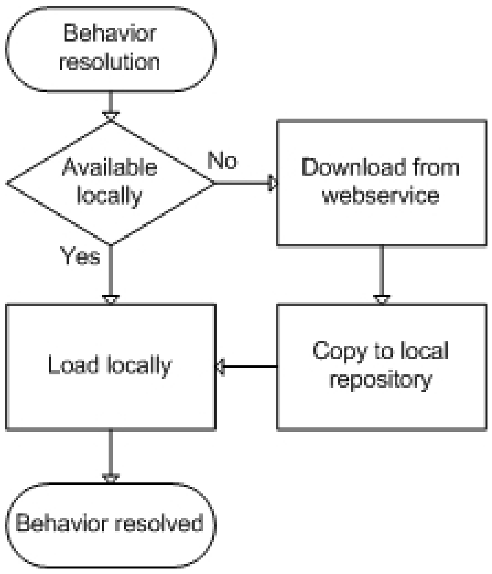


Figure 3.3: The Behavior Resolution process.

The extension will pass the name of the VE player and its version to the webservice, along with the GUID of the behavior it wants to resolve. If a behavior implementation for the VE player in question is available, the webservice will answer with the URI of that implementation and the Engine Extension can download it. More information about the Behavior Repository can be found in the next section.

The behavior implementations are essentially pieces of code, which are especially written to be used by the specific VE player and its platform. These implementations need to follow some specific guidelines, which depend on the VE player. These guidelines are essential if the Engine Extension needs to be able to interface with a behavior implementation dynamically. The next chapter will discuss these guidelines for an extension to the XNA Framework [17]. These guidelines can be seen as a set of rules which the code should follow. An example of such a guideline could be that the implementation contains a class which has the same name as the GUID and that each parameter can be set by invoking the method named “set”, appended with the name of the parameter and provided with the value as an argument (ex. setAngle(1.52)). These guidelines are specific to the VE player and provide the Engine Extension with a known interface which it can use to communicate with the behavior implementations.

Interface for VE player extensions

Because we want to minimize the required efforts for VE player developers to incorporate our approach into their products, we will now define an abstract interface by which the Extended Engine should be able to communicate with the VE player and the VE application. This interface will be defined as a set of functions. First of all, our Engine Extension should be able to load a Smart Entity file upon a request of the VE application. This is the **loadSmartEntity** function of our Engine Extension. It should load a Smart Entity file, by using its own **loadSmartEntityFile** function. This function is responsible for parsing the document and for instructing the VE player to visualize the related 3D model. This visualization is done by examining the Geometry Locators (see 3.2). The Engine Extension will select a compatible file format for the visual representation of the VO. The extension should then be able to request the VE player to load that representation so it can be rendered in the VE. This interface function will be named **loadGeometry**. Since our Engine Extension needs this loaded geometry in order to pass it to the behavior implementations, the VE player should also be able to return the geometry that has been loaded to the Engine Extension. This function will be named **returnGeometry**. When a VE application wants to invoke an ability of a VO, it will call the **invokeAbility** method of our Engine Extension. The Engine Extension should then execute the behaviors of the ability. These behaviors will modify the geometry of our VOs as a function of time, therefore the Engine Extension will need to call these behaviors periodically. To accomplish this, our Engine Extension should receive a synchronization signal from the VE player. Upon receiving this signal, the geometry of each VO can be updated in function of the behaviors that are being invoked and the execution time of these behaviors. All VE players already have this kind of synchronization internally, namely the game-loop [52]. This game-loop is called each time the scene needs to be redrawn. If we want the Engine Extension to update synchronously at the same rate, the VE player will need to send a notification to our Extended Engine, forcing it to update the VOs according to the behaviors which are currently being invoked. This signalization will be called the **update** function and should be called by the VE player within its game-loop. Obviously, the VE application will also want to make use of the information which has been annotated. By using the **requestInformation** function, the VE application can request the information of a specific Smart Entity. The Engine Extension can then communicate this information back to the VE application, which can visualize it appropriately.

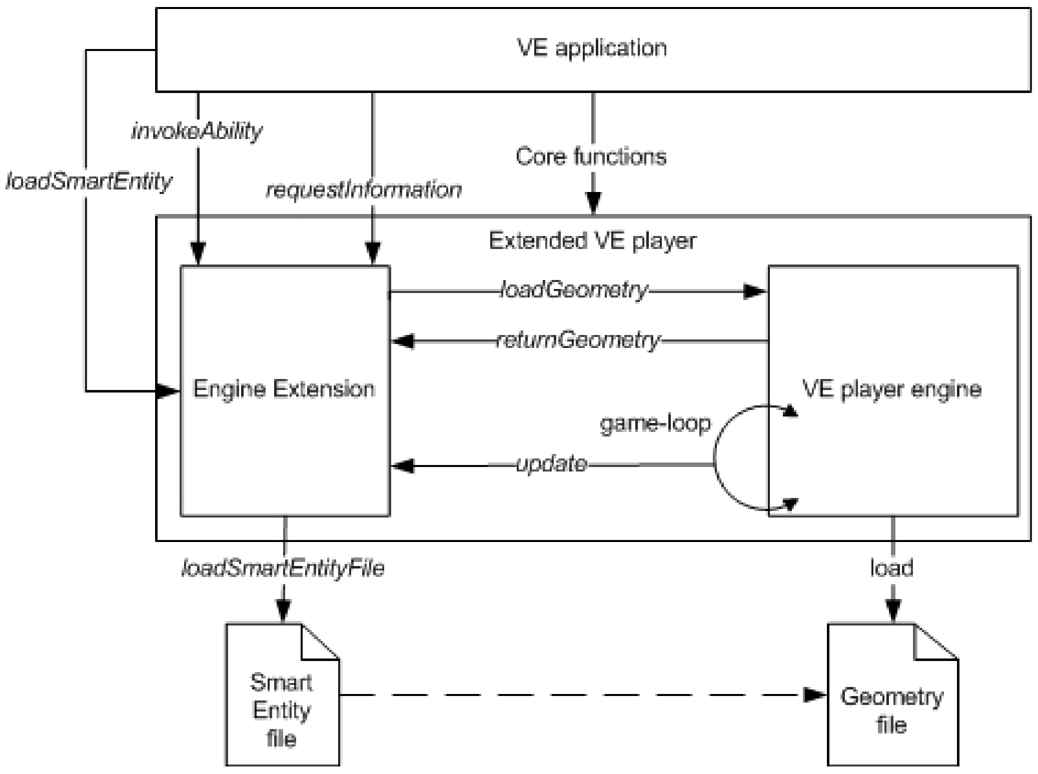


Figure 3.4: Conceptual overview of the interface.

To summarize, we have defined an abstract interface which defines how communication between the VE player, its Engine Extension and the VE application should occur (see figure 3.4). We have defined the following set of functions:

- **loadSmartEntity**: The Engine Extension can be asked to load a Smart Entity.
- **loadSmartEntityFile**: This function will parse the Smart Entity file into an internal representation and will call the *requestGeometry* function.
- **requestInformation**: The VE application can request information a Smart Entity, for instance its descriptors and available abilities. This information can then be displayed within the VE.
- **loadGeometry**: The VE player should provide a function which loads the actual geometry of the Smart Entity from a file which is compatible with the VE player. This file is chosen by the Engine Extension by examining the entity's Geometry Locators.
- **returnGeometry**: This function returns the geometry of the VO from the VE player to the Engine Extension, so the behaviors can update it.
- **invokeAbility**: This function can be called by the VE application if it needs to execute an ability of the Smart Entity.
- **update**: This function of the Extended Engine is called periodically by the VE player's game-loop. It will enumerate all of the abilities of the VO and their abilities which are being performed. For each ability, its behaviors will be updated in terms of the time the ability is running.

3.3.2 Behavior Repository

All of the platform-specific behavior implementations reside in a central repository. This is the Behavior Repository. Ideally, it holds references (URIs) to all implementations of behaviors for all different VE players, otherwise some behaviors cannot be invoked on certain VE players. The Engine Extension can query this repository because it can retrieve its address from the Behavior Locators within the Smart Entity file. These locators link behavior GUIDs to the URI of a Behavior Repository. When the Engine Extension queries the Behavior Repository for a specific behavior implementation, it sends its VE player code (ex. XNA, Java3D, O3D) and the version number of the VE player (ex. 3.0) along with the GUID of the behavior. The repository will respond with a URI to the behavior implementation for that VE player. Along with references to implementations, the repository also contains information about behaviors. This information is specific to each behavior GUID. As an example, consider the behavior with GUID “RotateLinear”. When queried with a GUID, the Behavior Repository can answer with the information it has about that behavior. This information contains a description of this GUID, a list of VE players for which it has a specific behavior implementation and the list of parameters this behavior requires, as well as a description per parameter and the types of the parameters.

This information can be used by authoring tools, like the one that will be presented in the next chapter, in order to help designers of VOs. Designers can then browse the repository, select a behavior they want to use and receive information on how they should use the behavior. For example, a designer wants to perform a rotation of the turret of a tank. The designer can then query the Behavior Repository for all of its available behaviors. The designer can select the RotateLinear behavior and use it. The authoring tool can then request additional information about the behavior in order to assist the designer. This way, the designer can see that the angle that is needed should be expressed in radians and that the rotation axis should be expressed as a vector in 3D space.

From the example that was given, we can deduce that the parameters of a behavior have types associated with them. In order to maintain portability, the values of these parameters should be expressed in an identical way for each type. The Engine Extension can then convert the values of the parameters to the types that are needed by the concrete Behavior Implementation.

Below is a list of the proposed data types of the parameters.

- **String**: Simple textual content, represented as text.
- **String[]**: An array of Strings, represented as each String separated by a semi-colon. This is the type of the required *TargetBoneIDs* parameter. Example: “part1;part2”
- **Number**: Represents a number, examples are the *Start* and *Duration* parameters.
- **Boolean**: Represents a boolean value. Represented as “True” or “False”
- **Vector3**: Represents a triple of numbers. Grouped within round brackets and separated by a comma. Example: “(1,2,3)”).

These types are used to indicate the type of the parameters to the Authoring Tool and to the user. The Authoring Tool can then provide an easy way of entering an array of strings or vectors. The Engine Extension will pass the values of the parameters to the behavior implementation. Since the value of the parameters are expressed as strings, it is then necessary to transform these strings to platform specific structures, such as XNA’s Vector3 class or Java3D’s Double class.

By centralizing all available behavior implementations and by providing a platform-independent interface by means of a webservice, several advantages have been achieved:

- **Increased behavior reusability**: Designers can make use of the same behavior implementation and the available behaviors can be browsed by a designer. This way, a designer only needs to specify the parameters of a behavior, while the concrete implementation has been done by programmers.
- **Increased portability**: The Behavior Repository provides a centralized point of access for all VE players on different platforms. By querying this repository, the Engine Extension can obtain the compatible behavior implementation. This way, Smart Entities could even be loaded at run-time by the VE player.
- **New behaviors can be added**: Because the Behavior Repository is an open medium, behavior implementations can be added, upon which they can be used by designers.
- **Existing behaviors can be updated**: Existing behaviors can be tuned to achieve a better performance or to eliminate bugs. To accomplish this, the Engine Extension could regularly check if updated implementations of the behaviors in its local repository are available.

Chapter 4

Implementation of the Authoring Tool and the Framework

The first part of this chapter will explain the Authoring Tool that has been developed in order to facilitate the creation of Smart Entities. The main features and the implementation of this tool will be discussed. The second part will elaborate on the Microsoft XNA Engine Extension that has been developed. The VE player that was chosen for this extension was Microsoft's XNA 3.0. The third and last part of this chapter will talk about the implementation of the Behavior Repository webservice.

4.1 Authoring Tool

4.1.1 General design

Obviously, we cannot expect designers to create and edit Smart Entities by altering the Smart Entity XML file format directly. In order to facilitate this task, an Authoring Tool has been developed which will enable designers to both create and edit the Smart Entity files in an attractive visual environment. The Authoring Tool that will be presented in this section is the prototype of a target-platform independent, Smart Entity editor. Meaning that the Smart Entity files which are created by using this Authoring Tool should not refer to specific behavior implementations but to the GUID that is attached to each behavior and to the Behavior Repository webservice on which the behavior with that GUID was found.

The Authoring tool provides the following functionalities:

1. **Create, read and edit Smart Entity files**: A user is able to create new Smart Entity files or can alter existing ones.

2. **Attach and detach Geometry Locators**: When a file is loaded, Geometry Locators (see 3.2) can be attached and detached.

3. **Visualization of the work**: Users can preview their work easily, resulting in a faster and more efficient production process.

4. **Manage descriptors**: Users are able to add descriptors to the Smart Entity by entering the key, value, type and visibility of the descriptor. They can also update or remove existing descriptors.

5. **Manage abilities**: Users can add abilities by providing a name. Abilities can also be removed or their name can be changed.

6. **Manage a set of available behaviors**: The application has its own repository of behaviors which contains the location of the Behavior Repository webservice where the behavior can be found and the parameters the behavior requires. The user can add behaviors to the application's repository or remove them.

7. **Manage behaviors per ability**: Users can select a behavior from the Behavior Repository, attach it to the ability and define the behavior's parameters (such as Start, Duration, TargetBoneIDs, etc.). Behaviors can also be updated or removed from the ability.

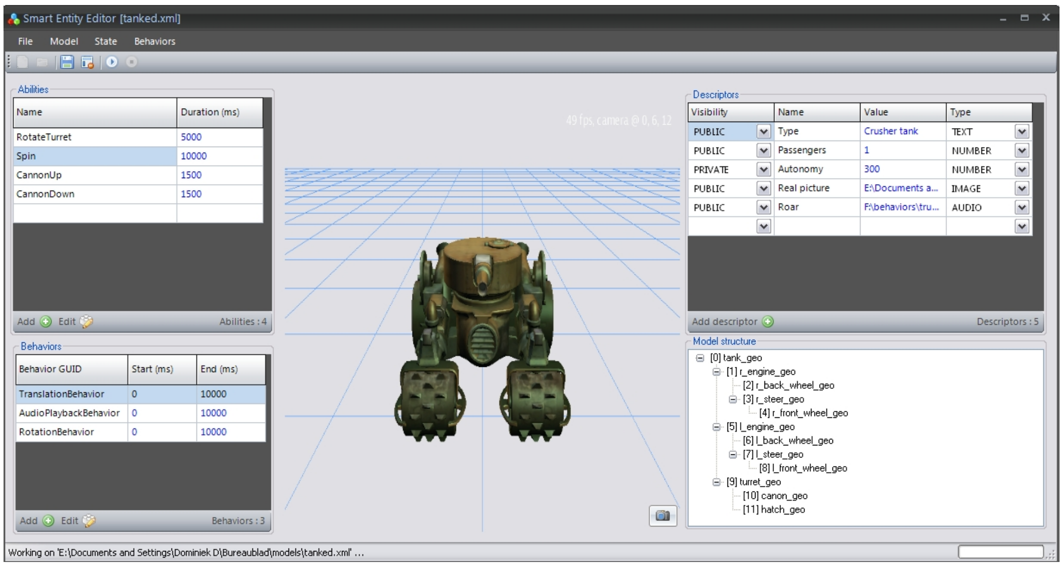
8. **Manage Behavior Locators behind the scenes**: When a new behavior is added to an ability, the application will update the Behavior Locators. The behavior's GUID will be added, along with the link to the Behavior Repository webservice where it can be found. If a behavior is removed (all instances of that behavior), the matching Behavior Locator is deleted.

9. **Manage State**: The user should be able to add state variables to an entity and assign an initial value to those variables.

10. **Manage Pre- and Postconditions**: The user is able to specify the conditions (state variables) under which an ability can be performed, i.e. the preconditions. User should also be able to define postconditions, which update the state of the entity when an ability has finished.

4.1.2 Implementation

Based on the set of features which has been discussed in the previous section, an application was developed using C#/.NET 3.5 and Microsoft Visual Studio 2008. This technology was chosen because it supports the design of user interfaces in an easy and intuitive way. By choosing the .NET framework, it was possible to use the same Smart Entity format parser for both the extension of the VE player and the Authoring Tool. For more details on the functionalities of the Authoring Tool, we refer the reader to chapter 5, where a walkthrough of the tool will be given during the Case Study. This Authoring Tool has been designed to provide an attractive and



Abilities

Name	Duration (ms)
RotateTurret	5000
Spin	10000
CannonUp	1500
CannonDown	1500

Add Edit Abilities : 4

Behaviors

Behavior GUID	Start (ms)	End (ms)
TranslationBehavior	0	10000
AudioPlaybackBehavior	0	10000
RotationBehavior	0	10000

Add Edit Behaviors : 3

Descriptors

Visibility	Name	Value	Type
PUBLIC	Type	Crusher tank	TEXT
PUBLIC	Passengers	1	NUMBER
PRIVATE	Autonomy	300	NUMBER
PUBLIC	Real picture	E:\Documents a...	IMAGE
PUBLIC	Roar	F:\behaviors\tru...	AUDIO

Add descriptor Descriptors : 5

Model structure

```
[0] tank_geo
    [1] r_engine_geo
        [2] r_back_wheel_geo
        [3] r_steer_geo
            [4] r_front_wheel_geo
    [5] l_engine_geo
        [6] l_back_wheel_geo
        [7] l_steer_geo
            [8] l_front_wheel_geo
    [9] turret_geo
        [10] canon_geo
        [11] hatch_geo
```

Working on 'E:\Documents and Settings\Dominiek D\Bureaublad\models\tanked.xml' ...

Figure 4.1: Annotating a Smart Entity by using the Authoring Tool.

efficient way of creating and editing Smart Entities. It contains a previewer which allows designers to visualize their work. This previewer uses the Smart Entity Engine, which will be discussed in the next section.

4.2 The XNA Engine Extension

This section will explain how we have created the Extended Engine for Microsoft's XNA 3.0 VE player [17], which has been called the "Smart Entity Engine". This technology was chosen for several reasons. First, XNA is well documented and supported by Microsoft. Therefore there is a sure future for this technology. Second, XNA is well accepted within the gaming industry and more and more games are being created with XNA. Third, by combining XNA and .NET, we have access to a large library of functionalities which ease the development of our application. Examples of these libraries are XNA's mathematics library and .NET's serialization functions [53]. XNA integrates well into Microsoft Visual Studio 2008 Professional which is one of the most productive and user-friendly Integrated Development Environments (IDE) available today. After we have discussed the extension, we will state which guidelines the created Engine Extension imposes on the behavior implementations.

4.2.1 XNA Smart Entity Engine

The XNA Smart Entity Engine is the implementation of an Engine Extension for Microsoft's XNA 3.0 VE player [17]. In XNA, loading custom content such as 3D model, sound or texture formats can be done by extending the Content Pipeline [41]. Because other VE players do not provide such architecture and because the created extension that will be discussed here will serve as a blueprint for creating other Engine Extensions for different VE players, we did not create such a Content Pipeline extension. In XNA, all VOs are represented internally by an instance of the *DrawableGameComponent* class. This class can be inherited from any new visual components the developer wants to add to the VW. Our Smart Entity engine has created such a specialization of this class, namely the *XNASmartEntity* class. An UML scheme of the architecture is displayed on the next page. When an instance of the *DrawableGameComponent* class is added to the game, its *update* and *draw* methods will be called continuously by the XNA framework. The *XNASmartEntity* class which has been created contains the geometry of the entity in an instance of XNA's *Model* class and contains an instance of the *SmartEntityDefinition* class, which contains the information from the Smart Entity file: descriptors and abilities. The VE can access and query this information and display it appropriately, as was discussed in chapter 3.

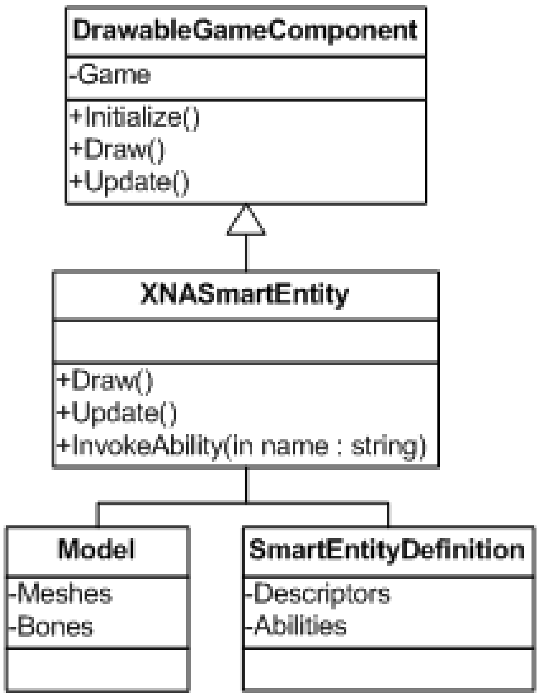


Figure 4.2: The Smart Entity in the XNA Framework.

Furthermore, the game can perform an ability of a Smart Entity by calling the *invokeAbility* method of the *XNASmartEntity* instance, in order to help explain how this is accomplished, the scheme display in figure 4.3 can be of help while reading.

When the *invokeAbility* method is called, the Smart Entity Engine will first check to see if the preconditions are met, given the current state of the object. This is done by comparing all of the variables values that were stated in the preconditions with the actual values of the entity's state (see chapter 3). When the preconditions are met, a new instance of the *XNAAbilityInstance* class will be created. Upon creation, the ability's definition will be iterated and for each behavior invocation that is used, an instance of *XNABehaviorInstance* will be added to the *BehaviorInstances* list. The *XNABehaviorInstance* will first construct an instance of the platform-specific behavior implementation (ex. *RotationBehavior*), which has been fetched during the Behavior Resolution (see 3.3), and will then set the instance's parameters according to those in the Smart Entity file. The values of the parameters in the definition will first be converted to XNA specific structures by using a type-conversion mechanism. This mecha-

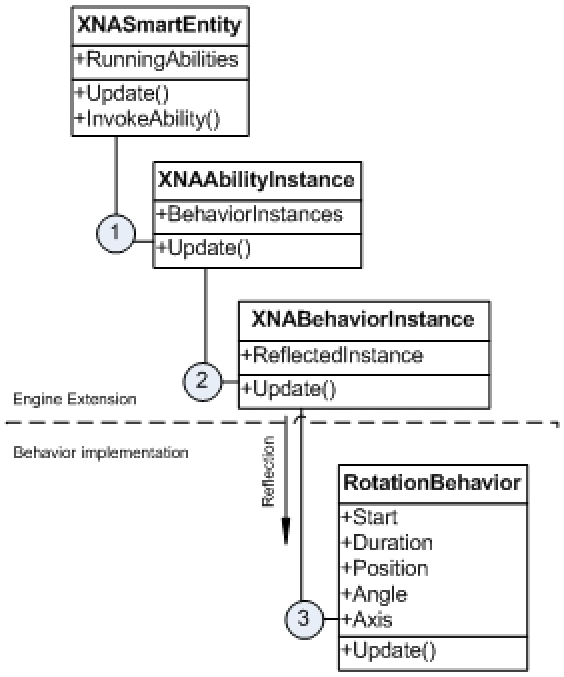


Figure 4.3: Propagation of updates in the game-loop.

nism first checks which datatype the behavior implementation expects and then attempts to transform the string value from the Smart Entity file's behavior invocation parameter to that type. When the *XNABehaviorInstance* instances have been created, the *XNAAbilityInstance* will be added to the collection of running abilities in the *XNASmartEntity* instance. Since all off the required behavior implementations have been resolved when the Smart Entity file was loaded, no more lookups need to occur and this process can be performed fast. During the game-loop, the *update* method of the *XNASmartEntity* instance will be called at regular intervals. During each call to *update*, the *XNASmartEntity* instance will loop through all of the abilities which are currently being performed (*RunningAbilities* member) and call the *update* method on those *XNAAbilityInstance* instances (1). Within this method, the *update* method of all *XNABehaviorInstance* instances in the *BehaviorInstances* member will be invoked (2). This *update* method will set the *Position* field of the dynamically loaded behavior implementation and will call the *update* method of the dynamic instance if necessary ($start \leq Position \leq start + duration$) (3). It is this instance that is responsible for updating the model's geometry and actually performing the behavior. More information on the way these behaviors are implemented can be found in the next sections. When an ability has finished, so when all of the behavior instances have finished, the Smart Entity engine will set the state variables of the entity according to the post-conditions of the Smart Entity, thereby updating the state the entity is in.

4.2.2 Smart Entity engine interface

In chapter 3, we have proposed an interface for the VE player extensions. In this section, we will provide an overview of how our XNA Smart Entity engine implements this interface.

- **loadSmartEntity**: The VE application which uses our XNA Smart Entity can request our Smart Entity engine to load a Smart Entity file. The engine will then load the information from the file and create a new instance of the *XNASmartEntity* class. This instance will then be returned.
- **loadSmartEntityFile**: This function will be called by the Smart Engine extension and will parse our Smart Entity file into an object-oriented representation. We have created a parser which is decoupled from the Microsoft XNA framework and which could be used by other .NET based applications (like our Authoring Tool).
- **requestInformation**: All of the information from our Smart Entity file is available within an instance of the *XNASmartEntity* class. This instance has been returned by the *loadSmartEntity* function.
- **loadGeometry**: Our Smart Entity engine will enumerate all Geometry Locators of the Smart Entity file and will ask the XNA framework to load either a DirectX or FBX file, depending on which one is available.
- **returnGeometry**: The XNA framework will return an instance of its *Model* class, which holds the necessary geometry and which is encapsulated in the *XNASmartEntity* instance.
- **invokeAbility**: The *XNASmartEntity* instance provides a method which can invoke an ability by passing its name as an argument. This process is described in more detail in 4.2.1.
- **update**: In XNA, adding code to the game-loop is easy. By inheriting from the *DrawableGameComponent* class, implementing the *Draw* and *Update* methods and adding the instance to the game's *Components* attribute the *Draw* and *Update* methods will be called

periodically. For more information on this implementation, please see 4.2.1.

4.2.3 Behavior implementation guidelines

Before any Engine Extension can load code at runtime by using reflection techniques [51], the code in question should be created with some guidelines in mind. These guidelines can be seen as the specification of an interface between the Engine Extension and the behavior implementations. This interface allows these two components to interact with each other. For our XNA Smart Entity engine and its behaviors, we have stated the following guidelines:

1. The file which contains the platform-specific implementation has the format “guid.dll”, in which guid represents the GUID of the behavior. (ex. rotationbehavior.dll)

2. The name of the class which implements the behavior is the same as the GUID.

3. The constructor of that class accepts two parameters: an instance of the *Game* class, which is an XNA specific class that contains information on the rendering and context. The second parameter is the *Model* class. This class is also specific to XNA and contains the geometric data of the model: its bones and transformation matrices. It is important to know that these matrices are passed by reference.

4. The parameters are applied to the behavior implementation instance by setting fields. These fields are the way to define data accessors in C# [54]

5. The behavior implementation contains an update method, which takes no parameters. This method will be called when the behavior needs to be invoked. It is within this method that the actual modification of the geometry in function of time can take place.

6. Before the update method is called, the Smart Entity Engine will set the Position field of the behavior implementation to the elapsed time (in milliseconds), since the ability was invoked.

When a behavior implementation is created with these guidelines in mind, our Smart Entity Engine is able to load the behavior dynamically and can interface with it. To facilitate the creation of new behavior implementations, a template class has been designed which can be inherited from. This code, as well as the code for the TranslationBehavior and RotationBehavior can be found in the appendix (8.4). The guidelines that were stated were specifically conceived for our Smart Entity engine, but they can be generalized towards other object-oriented environments, such as Java or C++.

4.3 Behavior Repository

The Behavior Repository is implemented as an ASP.NET 3.5 webservice (.asmx). Using a webservice promotes the interoperability we wish to strive for. The webservice exposes its capabilities by using a WSDL (Webservice Description Language) scheme and uses SOAP (Simple Object Access Protocol) to transfer its information to clients (an Engine Extension or an Authoring Tool), both of these standards are supported across most of the platforms that are in use today The information on the behaviors themselves is stored using a plain XML file that has the following structure:

```
<Behavior guid="RotationBehavior"
description="Perform a linear rotation">
 <Parameters>
  <Parameter name="TargetBoneIDs" type="String[]"
       description="" />
  <Parameter name="Duration" type="Number"
       description="" />
  <Parameter name="Start" type="Number"
       description="" />
  <Parameter name="Angle" type="Number"
       description="The angle in radians" />
  <Parameter name="Axis" type="Vector3"
       description="The axis of rotation" />
 </Parameters>
</Behavior>

<Player code="Java3D" version="1.3">
  <BehaviorLocation guid="RotationBehavior"
   uri="http://.../java3d/13/RotationBehavior.jar" />
</Player>
<Player code="XNA" version="2.0">
  <BehaviorLocation guid="RotationBehavior"
   uri="http://.../xna/20/RotationBehavior.dll" />
</Player>
```

The Behavior Repository webservice exposes three web-methods:

- **DescribeBehavior(*guid*)**: Returns the description of the behavior identified by the GUID parameter, independent of the platform. This is the description of the behavior, along with the parameters that are needed, their descriptions and their datatypes.
- **FindBehavior(*guid, playerCode, playerVersion*)**: This method is to be invoked by the Engine Extension. The extension will contact the webservice with the GUID of the behavior it needs to invoke, the playerCode (name of the VE player) and the version of the player. The repository will respond with the description of the behavior and a URI for the behavior implementation (code file) for that VE player, if available.
- **ListAvailableBehaviors()**: Returns a list of descriptions for all the behaviors that are available in the Behavior Repository. This method is mainly used by the Authoring Tool to provide means of browsing and exploring the Behavior Repository to the designer.

Upon a request, the XML file is parsed by the Behavior Repository webservice and the data-structures (defined in the WSDL) are passed on to the client using SOAP.

Chapter 5

Case study

To demonstrate the approach that was presented in this work and the techniques that were developed, a case study will be presented.

5.1 Outline

In this case study, we will be creating a VE application and its content from scratch. More concretely, we will develop an application that will display a cellphone in a 3D environment. The main purpose of this environment is promotion. As several research papers have shown, a company can benefit greatly from these promotional 3D environments (see [55], [56] [57]). We want to deliver the annotations (information) which we will add to the VO of the cellphone by using the Authoring Tool and allow some basic interaction with the VO. Furthermore, it should be possible to reuse the environment for different cellphones and we should be able to reuse the cellphone on different VE players like Google's O3D [36]. We will model the 3D representation, create a Smart Entity file (see 3.2) by using the presented Authoring Tool(see 4.1), use the Smart Entity Engine (see 4.2) to load and use this file and finally create the VE application itself. In order to deliver a rich user experience, we will also add some requirements to the environment:

1. Users should be able to view the cellphone from different perspectives, the cellphone can be rotated and zoomed into.
2. The specifications of the cellphone should be displayed in the top-left corner.
3. Images should be shown at the right of the screen.

4. The cellphone can be opened and closed by the user, depending of its state (opened or closed).

5.2 3D Modeling

The cellphone we will use in this case study is the Nokia 5300 [58]. It has been modeled using AutoDesk's 3ds Max [7] and exported to the COLLADA [20] format by using the built-in COLLADA exporter and to the DirectX (X) format with the Panda DirectX Exporter for 3ds Max [59]. To

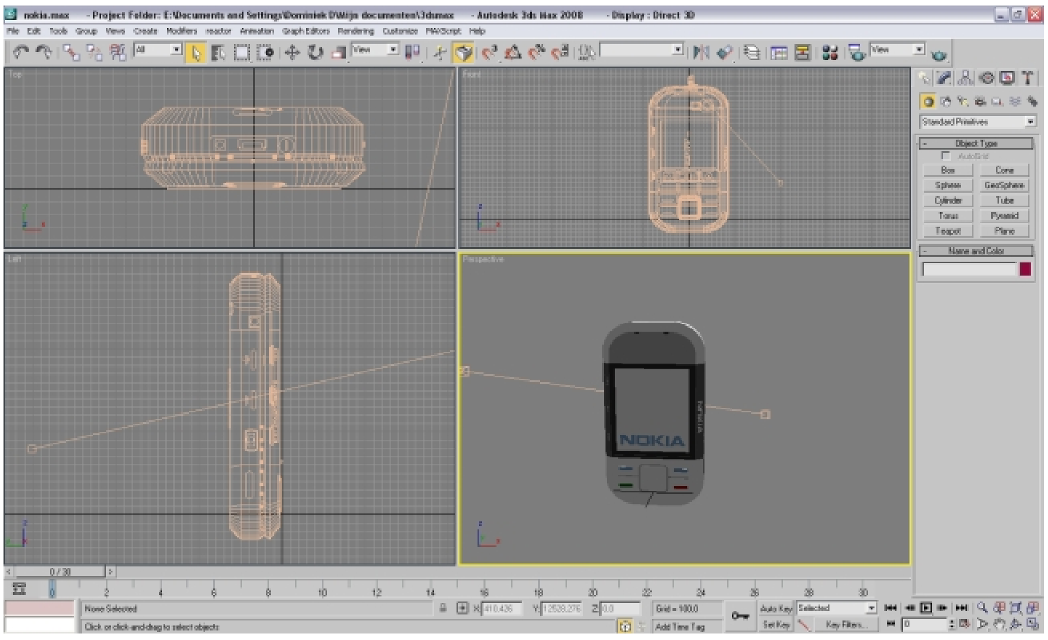
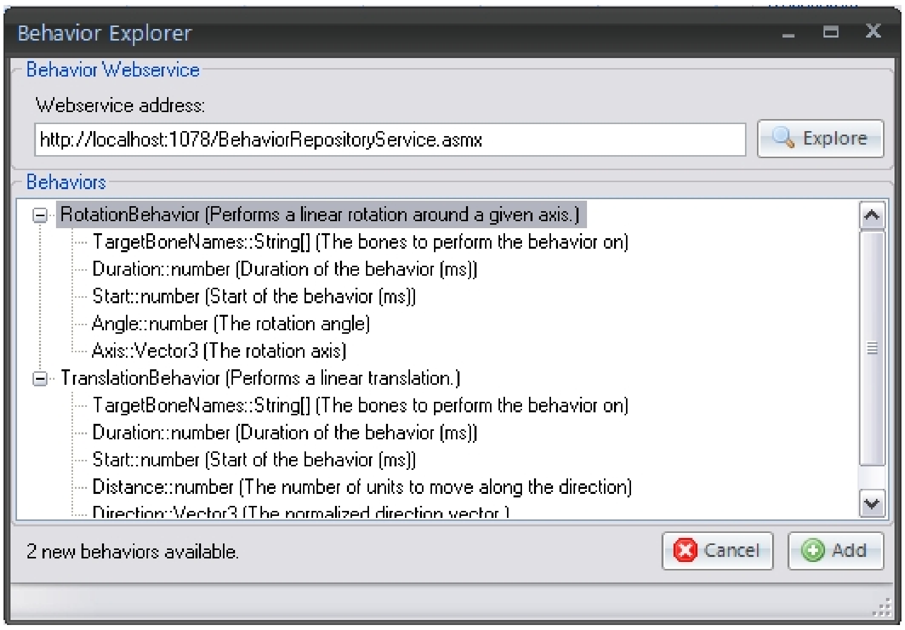

nokia.max - Project Folder: E:\Documents and Settings\Dominiek D\Mijn documenten\3dsmax - Autodesk 3ds Max 2008 - Display : Direct 3D
File Edit Tools Group Views Create Modifiers reactor Animation Graph Editors Rendering Customize MAXScript Help
Top
Front
Left
Perspective
NOKIA
Standard Primitives
Object Type
AutoGrid
Box
Cone
Sphere
GeoSphere
Cylinder
Tube
Torus
Pyramid
Teapot
Plane
Name and Color
0 / 30
None Selected
Click or click-and-drag to select objects
Grid = 100,0
Add Time Tag
Auto Key
Selected
Set Key
Key Filters...

Figure 5.1: The Nokia 5300 being modeled in 3ds Max.

allow the open and close animation, we need to name the parts which make up the model's front appropriately. This can be accomplished in 3ds Max by selecting the meshes with the mouse and simply setting the name in the panel on the right. We will then be able to reference these in the *TargetBoneIDs* parameter of our behaviors, so they can be animated.

5.3 Creating the Smart Entity file

To create the Smart Entity file, we will use the presented Authoring Tool (see 4.1). We start by creating a new Smart Entity file. We will first attach the cellphone models (COLLADA and DirectX) to the file. The previewer will then visualize the model (DirectX format) in the Authoring Tool. Then we can start annotating our Smart Entity by adding the descriptors. The descriptors can be seen in the right upper corner of the application. Five textual descriptors and four images of the cellphone have been added. In order to add abilities to the entity, we will need to have some available behaviors to work with. For this, we will use the Authoring Tool's Behavior Explorer to browse a Behavior Repository webservice. The Behavior



Webservice address:
http://localhost:1078/BehaviorRepositoryService.asmx

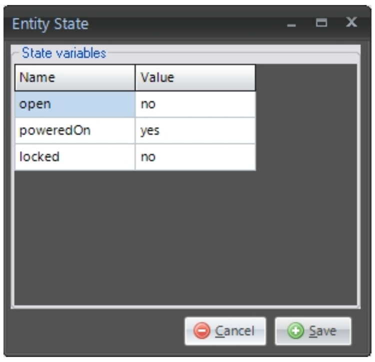
RotationBehavior (Performs a linear rotation around a given axis.)
TargetBoneNames::String[] (The bones to perform the behavior on)
Duration::number (Duration of the behavior (ms))
Start::number (Start of the behavior (ms))
Angle::number (The rotation angle)
Axis::Vector3 (The rotation axis)
TranslationBehavior (Performs a linear translation.)
TargetBoneNames::String[] (The bones to perform the behavior on)
Duration::number (Duration of the behavior (ms))
Start::number (Start of the behavior (ms))
Distance::number (The number of units to move along the direction)
Direction::Vector3 (The normalized direction vector.)

2 new behaviors available.

Figure 5.2: The Authoring Tool's Behavior Explorer.

Explorer allows users to add a behavior they found to Authoring Tool's toolbox of available behaviors. This toolbox is maintained by the Authoring Tool itself and allows reuse of behaviors. Using the Behavior Explorer, we have added both the translation- and rotation-behaviors to the toolbox. The XML file which has been created by the Authoring Tool, can be seen in the Appendix (see 8.2)).

Because we do not want to be able to open the cellphone twice, we will attach some state to our Smart Entity. The Authoring Tool allows us to attach state variables to the entity. This functionality is displayed below: We have added and initialized three state variables to our Smart Entity. We

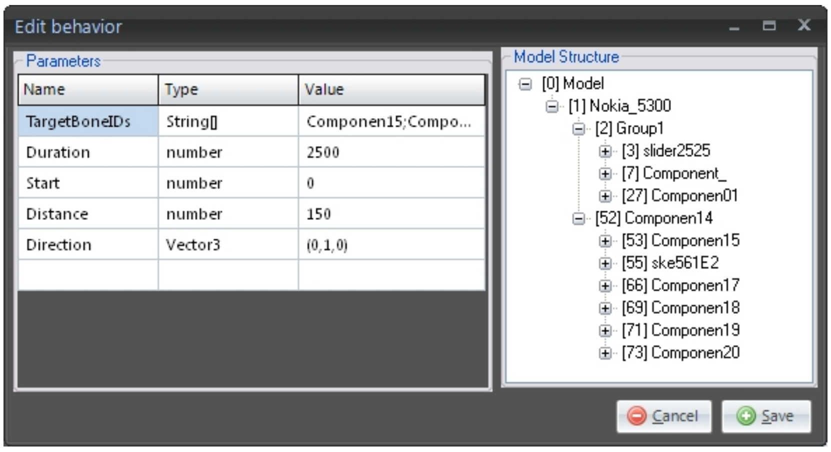


State variables

Name	Value
open	no
poweredOn	yes
locked	no

Figure 5.3: Adding state variables in the Authoring Tool.

can now go on and define our “open” and “close” abilities. When we add an ability, we first need to give it at name. Second, we need to add a set of behaviors to the ability. Because opening the cellphone actually consists of a translation of the front of the cellphone, we will use the translation-behavior which we have already added to the Authoring Tool’s toolbox. All we need to do now is parameterize this behavior, this process is shown below: This is the only behavior we need to add to the “Open” ability. The



Parameters

Name	Type	Value
TargetBoneIDs	String[]	Componen15;Compo...
Duration	number	2500
Start	number	0
Distance	number	150
Direction	Vector3	(0,1,0)

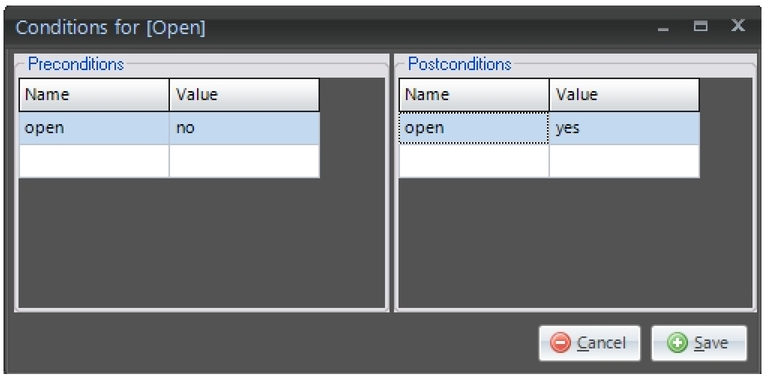
Model Structure

```
⊟ [0] Model
  ⊟ [1] Nokia_5300
    ⊟ [2] Group1
      ⊞ [3] slider2525
      ⊞ [7] Component_
      ⊞ [27] Componen01
    ⊟ [52] Componen14
      ⊞ [53] Componen15
      ⊞ [55] ske561E2
      ⊞ [66] Componen17
      ⊞ [69] Componen18
      ⊞ [71] Componen19
      ⊞ [73] Componen20
```

Figure 5.4: Specifying parameters of a behavior invocation.

“Close” ability is analogous to the “Open” ability, except for the translation direction vector which should be $(0, -1, 0)$.

When this is done, we can specify the pre- and postconditions of the abilities. This process is shown below. The precondition that is depicted


Conditions for [Open]

Preconditions

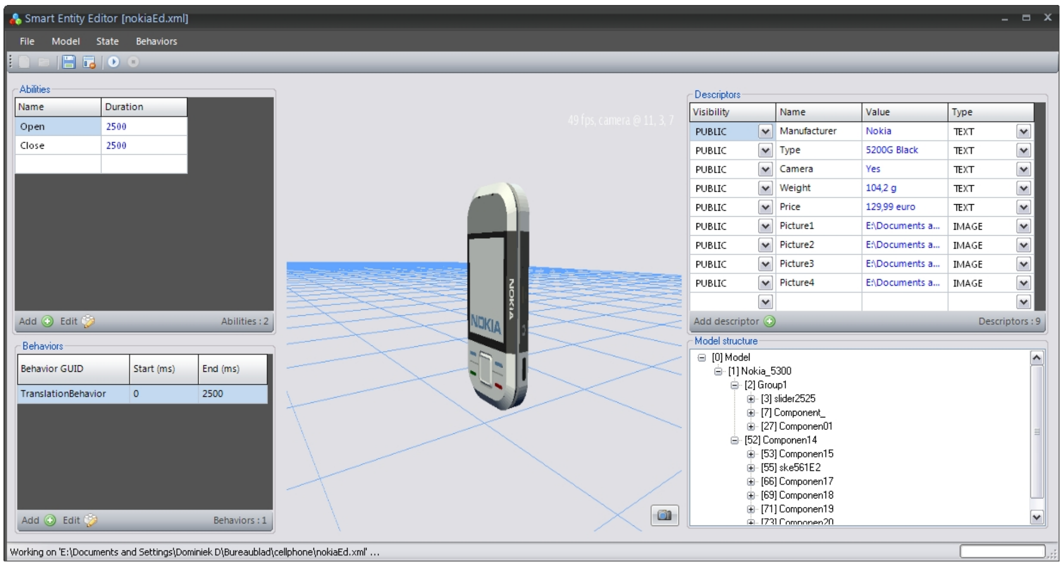
Name	Value
open	no

Postconditions

Name	Value
open	yes

Figure 5.5: Specifying post- and preconditions for the “Open” behavior.

states that the cellphone cannot be opened when it is already opened and that the “open” state variable’s value should change to “yes” when the ability was performed. This concludes the creation of the Smart Entity file by using the Authoring Tool, we can now save this file to disk and our VE application will be able to load it.


Smart Entity Editor [nokiaEd.xml]

Abilities

Name	Duration
Open	2500
Close	2500

Behaviors

Behavior GUID	Start (ms)	End (ms)
TranslationBehavior	0	2500

Descriptors

Visibility	Name	Value	Type
PUBLIC	Manufacturer	Nokia	TEXT
PUBLIC	Type	5200G Black	TEXT
PUBLIC	Camera	Yes	TEXT
PUBLIC	Weight	104,2 g	TEXT
PUBLIC	Price	129,99 euro	TEXT
PUBLIC	Picture1	E:\Documents a...	IMAGE
PUBLIC	Picture2	E:\Documents a...	IMAGE
PUBLIC	Picture3	E:\Documents a...	IMAGE
PUBLIC	Picture4	E:\Documents a...	IMAGE

Model structure

```
[0] Model
  [1] Nokia_5300
    [2] Group1
      [3] slider2525
      [7] Component_
      [27] Componen01
    [52] Componen14
      [53] Componen15
      [55] ske561E2
      [66] Componen17
      [69] Componen18
      [71] Componen19
      [73] Componen20
```

Working on 'E:\Documents and Settings\Dominiek D\Bureaublad\cellphone\nokiaEd.xml' ...

Figure 5.6: The main window of the Authoring Tool.

5.4 VE design and programming

To develop a VE application which uses the Smart Entity file which we have created, we have used the extension to the XNA VE player which has been presented in chapter 4, the Smart Entity Engine. To recapture: the VE application is responsible for the logic of our application (rules, menus, content, etc.) and uses the VE player to help visualize and control the VE. Note that if we would want to develop our environment using another VE player such as one based on Java3D [16], we would need to use an Engine Extension for that VE player. The Smart Entity file will remain the same, so it is a platform-independent specification of the VO and its abilities. The Smart Entity engine (see 4.2) is able to read and visualize the Smart Entity and can invoke the abilities of our cellphone (open and close).

We will now explain how the VE application has been created by using the Smart Entity engine and the Smart Entity file of the cellphone. We will use Microsoft Visual Studio 2008 Professional edition to develop this environment. First off all, we have created a new XNA 3.0 game. Then we add a reference to the Smart Entity engine. Before we can see anything in the world, we need to configure the camera system (see 2.1). The camera needs to react to user input: we want to be able to rotate around the cellphone in any direction (up, down, left, right) and zoom in and out. The implementation of this camera system was a good exercise in vector algebra which is out of the scope of this thesis, but the implementation can be found in the appendix (see 8.3). Now that we have our camera system available, we can start making the environment. We will start by loading the Smart Entity and placing in the environment, this requires the following piece of code:

```
protected void InitializeRequestedSmartEntity()
{
  SmartEntityDefinition sed = SEDParser.Instance.Load(
   @"E:\nokia.xml");
  CellphoneModel cellphone = new CellphoneModel(this, sed);
  Quaternion rot = Quaternion.CreateFromAxisAngle(
   Vector3.Normalize(
   new Vector3(-0.2f, -0.4f, 0.15f)),
   MathHelper.PiOver4 / 2);
  cellphone.Rotation = Matrix.CreateFromQuaternion(rot);
  this.Components.Add(se);
}
```

First, the parser will load the Smart Entity file which we created using our Authoring Tool. The *CellphoneModel* class, which inherits from the *XNASmartEntity* class will be instantiated. The constructor will select an appropriate geometry format, in this case the X format, from our definition and will load it into XNA's *Model* class. To have a nicer visualization, the model is rotated slightly and finally our *CellphoneModel* instance is added to the *Game*'s *Component* property so that it will be drawn and updated in the game-loop. Next thing we want to do is to process the Smart Entity's descriptors and display them in the VE. We will display all public descriptors of type *Text* in the left upper corner. An example would be “Manufacturer:Nokia”. We will also display all public descriptors of type *Image*. These will be positioned on the right of the display. The code that is needed to accomplish this is display below:

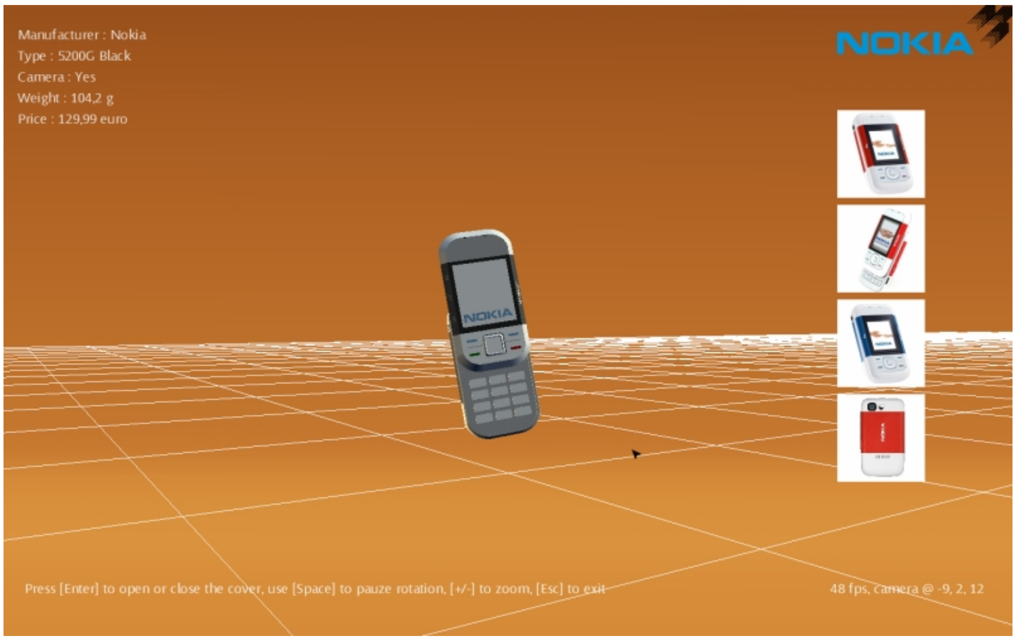
```
public void DisplayImages() {
 Vector2 position = new Vector2(
                        this.Window.ClientBounds.Width - 250, 150);

 foreach (DescriptorDefinition imgDesc in
             Cellphone.Definition.SelectDescriptors(
               VisiblilityTypes.PUBLIC, ContentTypes.IMAGE))
 {
  ImageOverlay overlay = new ImageOverlay(this,
                                          spriteBatch,
                                          imgDesc.Value,
                                          position);
  position += new Vector2(0, 135);
  Components.Add(overlay);
 }
}
```

The descriptors collection can be queried by providing two parameters: the visibility and the type of the descriptors. For each public image descriptor, an *ImageOverlay* instance will be added to the scene's components. This *ImageOverlay* class will draw this image on the right of the screen, at the offset specified by the *position* variable. Another requirement was the display of the cellphone's specifications in the top left corner of the screen. This is accomplished in about the same way as the images. Instead of images, we will query for text and instead of the *ImageOverlay* instance, we will add a *TextOverlay* to the *Components* property. Using the Authoring Tool, we have created two abilities: “open” and close. If a user presses the Enter key, we want the cellphone to open if closed and vice versa. In order to make this possible, we will add a *Toggle* method to our *CellphoneModel* class. Within this method, we will check the state of the entity and start the appropriate ability.

```
public void Toggle()
{
 if (IsOpen())
  InvokeAbility("close");
 else
  InvokeAbility("open");
}
```

That is it for the VE design and programming part. We can now visualize our environment.

5.5 Result


Manufacturer : Nokia
Type : 5200G Black
Camera : Yes
Weight : 104,2 g
Price : 129,99 euro

Press [Enter] to open or close the cover, use [Space] to pauze rotation, [+/-] to zoom, [Esc] to exit

48 fps, camera @ -9, 2, 12

Figure 5.7: The opened cellphone in our created environment.

Above, a screenshot of the created environment is shown. Our Authoring Tool (see 4.1) has been used for the creation of our Smart Entity. This tool provides an easy to use front-end so that anyone can create new or alter existing Smart Entity files. Furthermore, the Authoring Tool has allowed us to browse a Behavior Repository (see 4.3) and has let us use the available behaviors in our Smart Entity file. We have also created an attractive virtual promotion environment. By using the Smart Entity engine (see 4.2), we were able to use our created Smart Entity file. We have used the typed descriptors (images and text) and have relied on the XNA Smart Entity engine to perform abilities (open, close) for us, simply by executing the *invokeAbility* method. The approach we have taken has given us some benefits. For instance, we can still change our Smart Entity file and the changes will be reflected in our VE without recompilation. We could add specifications, add images, create new abilities, etc. We can also reuse our Smart Entity file in other virtual environments, without changing it, and even on different VE players, if an Engine Extension is available for that player. The entire scene could also be reused for other cellphones, requiring none or minimal changes to the code of the VE application.

Chapter 6

Future work

The work that has been presented in this thesis is far from being complete, therefore this chapter will suggest a number of possible improvements.

Allowing Smart Entity interaction.

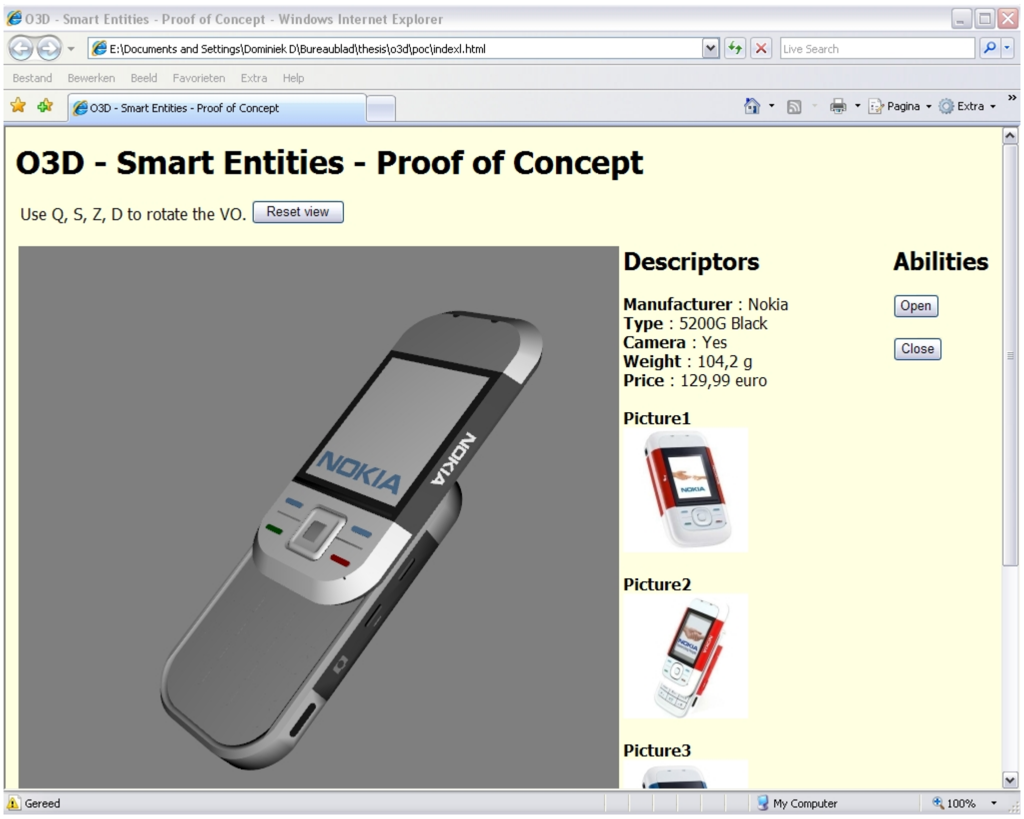
The current approach does not enable interaction between Smart Entities in the VE. For instance, when one entity collides with the other, the first could change color. Adding these kinds of triggers to our Smart Entity format would not pose a problem, but in order to use these triggers the interface of the Engine Extension, which was presented in 3.3 should be extended. The VE player could monitor events in the VE (collisions, mouse and keyboard interaction, etc.) and then trigger an event function on the Extended Engine, ex. *eventOccured*. The VE player could then decide what should be done, depending on the Smart Entities which triggered the event.

Improving the Authoring Tool

As with any application, the Authoring Tool can always be improved further. The current version of the Authoring Tool depends on the user to choose the right sequence of behaviors. A more advanced tool could use Constraint Solving techniques [60] to calculate the correct sequence and parameters. The functionalities of the tool could also be improved, for instance the abilities could be shown as a timeline. This would provide a better overview of the ability to the user. The tool could also provide more help to users while they adjust the parameters of a behavior. Some aid could be given when the user has to enter an angle or a vector, which would also lower the level of knowledge that is needed in order to work with the tool.

Creating an Engine Extension for another VE player

In this work, we have created a concrete implementation of our proposed Engine Extension for Microsoft's XNA framework. In order to fully test interoperability, another extension for an existing VE player could be made. A proof of concept Engine Extension the uses Google's O3D framework [36] is being developed at the time of writing, but since this technology was only released by the end of April and because large parts of the documentation are not yet available, this second Engine Extension could not yet be presented in this work. A screenshot of the proof of concept can be seen in figure 6.1.


O3D - Smart Entities - Proof of Concept - Windows Internet Explorer
E:\Documents and Settings\Dominiek D\Bureaublad\thesis\o3d\poc\indexl.html
Live Search
Bestand Bewerken Beeld Favorieten Extra Help
O3D - Smart Entities - Proof of Concept
Pagina Extra
Reset view
Open
Close
Gereed
My Computer
100%

O3D - Smart Entities - Proof of Concept

Use Q, S, Z, D to rotate the VO.

Descriptors

Manufacturer : Nokia
Type : 5200G Black
Camera : Yes
Weight : 104,2 g
Price : 129,99 euro

Picture1

Picture2

Picture3

Abilities

Figure 6.1: A limited prototype of the approach using Google O3D.

Chapter 7

Conclusion

Currently, the annotations that are supported by existing authoring tools are often text based and they are only related to the static part of a VO. Our approach has shown that it is possible to also annotate the dynamic part of a VO i.e. its behaviors and interactions. This was the first aim of the thesis.

By annotating behaviors and by knowing the behaviors that are associated to a VO, we were also capable of stating what a VO can do. This has allowed us to tell the user what a VO is capable of (i.e. what its abilities are).

In our approach, we have developed a new kind of file format, which is dedicated to these annotations. This file format has been called the Smart Entity file format.

We also made the observation that the portability across VE players is not well supported. By portability, we meant that VOs and their behaviors may not run on different VE players. Part of the reason for this is that, although VE players support different geometry file formats (X3D [19], COLLADA [20], etc.), the behaviors are still encoded into specific languages which only a group of the VE players can interpret and use. As a result, the VO is not very portable across VE players and often needs to be changed in order to run on a different VE player. This increases the cost of producing a VE and is also very frustrating and time-consuming for the designer. For these reasons, our approach has introduced a framework that will improve the portability of VOs across multiple VE players by using our Smart Entity file format and by assuming that developers of VE players will be willing to extend the VE players if they understand the benefits and if the necessary extension is kept small. To address this second aim, we have created a framework which facilitates the portability of VOs across different VE players.

The thesis has also produced an Authoring Tool which allows designers to generate Smart Entity files related to a VO. A case study was also provided which made use of the presented approach and its implementations. Nevertheless the framework has not been fully tested as we still need to show that portability of a VO can happen across several VE players.

Bibliography

[1] D. Livinstone and J. Kemp. Integrating web-based and 3d learning environments second life meets moodle. *UPGRADE*, 9(3):8–14, 2008.

[2] http://www.ohsu.edu/ohsuedu/newspub/releases/062005virtual.cfm. New Virtual Reality Surgery Simulator Hones Surgeons' Skills, Improves Patient Safety. Last accessed May 28, 2009.

[3] L. Chittaro and R. Ranon. Serious games for training occupants of a building in personal fire safety skills. *Proceedings of VS-GAMES'09: IEEE First International Conference on Games and Virtual Worlds for Serious Applications*, pages 76–83, 2009.

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

[5] http://www.psychsci.manchester.ac.uk/research/projects/intrepid. INTREPID Project. Last accessed May 28, 2009.

[6] J. Vince. *Introduction to Virtual Reality*. Springer, 1st edition edition, 2004.

[7] http://usa.autodesk.com/adsk/servlet/index?siteID=123112id= 5659302. Autodesk 3ds Max. Last accessed May 28, 2009.

[8] http://sketchup.google.com. Google Sketchup. Last accessed May 28, 2009.

[9] O. De Troyer, F. Kleinermann, B. Pellens, and W. Bille. Conceptual modeling for virtual reality. In *ER '07: Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling*, pages 3–18, Darlinghurst, Australia, Australia, 2007.

[10] T. Gutschmidth. *Game programming with Python, LUA and Ruby*. Course Technology PTR, 2003.

[11] A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, 1st edition edition, 2002.

[12] B. Pellens, F. Kleinermann, O. De Troyer, and W. Bille. Model-Based Design of Virtual Environment Behavior. *Proceedings of the 12th International Conference on Virtual Systems and Multimedia.*, pages 29–39, 2006.

[13] http://www.secondlife.com/. Second Life. Last accessed May 28, 2009.

[14] M. Slater, A. Steed, and Y. Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real-Time*. Addison Wesley, 2001.

[15] http://usa.autodesk.com/adsk/servlet/index?id=7635018&siteID=123112. Autodesk Maya. Last accessed May 28, 2009.

[16] http://java.sun.com/javase/technologies/desktop/java3d/. Java 3D API. Last accessed May 28, 2009.

[17] http://msdn.microsoft.com/en us/xna/. Microsoft XNA. Last accessed May 28, 2009.

[18] http://developer.vivaty.com. Vivaty Player and Vivaty Studio. Last accessed May 28, 2009.

[19] http://www.web3d.org. The Web3D Consortium. Last accessed May 28, 2009.

[20] http://www.khronos.org/collada. The Khronos Group on Collada. Last accessed May 28, 2009.

[21] http://www.research.philips.com/technologies/projects/3ddisp.html. 3D - The next revolution in TV viewing. Last accessed May 28, 2009.

[22] http://www.merl.com/projects/3dtv. Mitsubishi 3DTV. Last accessed May 28, 2009.

[23] http://www.evl.uic.edu/pape/CAVE/oldCAVE/CAVE.overview.html. CAVE Automatic Virtual Environment Overview. Last accessed May 28, 2009.

[24] http://graphics.cs.brown.edu/research/adviser/. ADVISER PSE (Advanced Visualization in Solar System Exploration and Research Problem Solving Environment). Last accessed May 28, 2009.

[25] http://vis.cs.brown.edu/areas/projects/cavepainting.html. CAVE Painting. Last accessed May 28, 2009.

[26] http://planetjeff.net/ut/CaveUT.html. CaveUT2004. Last accessed May 28, 2009.

[27] http://www.mobilizy.com/wikitude.php. Mobilizy's WikiTude Augmented Reality. Last accessed May 28, 2009.

[28] http://www.web3d.org. Web3D Consortium. Last accessed May 28, 2009.

[29] http://www.web3d.org/x3d/vrml. Virtual Reality Modeling Language (VRML). Last accessed May 28, 2009.

[30] http://www.octaga.com. Octaga Player. Last accessed May 28, 2009.

[31] http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=6837478. Autodesk's FBX 3D Model format. Last accessed May 28, 2009.

[32] http://www.blender3d.org/cms/Home.2.0.html. Blender. Last accessed May 28, 2009.

[33] http://www.adobe.com/products/playercensus/flashplayer. Flash Player Penetration. Last accessed May 28, 2009.

[34] http://www.papervision3d.org. Papervision3D. Last accessed May 28, 2009.

[35] http://www.alternativaplatform.com. Alternativa3D. Last accessed May 28, 2009.

[36] http://code.google.com/apis/o3d. Google O3D. Last accessed May 28, 2009.

[37] http://msdn.microsoft.com/en us/library/bb200104.aspx. Microsoft XNA Official Documentation. Last accessed May 28, 2009.

[38] http://creators.xna.com. Microsoft XNA Creators community. Last accessed May 28, 2009.

[39] R. Grootjans. *XNA 2.0 Game Programming Recipes: A Problem-Solution Approach*. Apress, 2008.

[40] C. Carter. *Microsoft XNA Unleashed: Graphics and Game Programming for Xbox 360 and Windows*. Sams, 2007.

[41] http://msdn.microsoft.com/en us/library/bb203887.aspx. The XNA Content Pipeline. Last accessed May 28, 2009.

[42] F. Kleinermann, O. De Troyer, C. Creelle, and B. Pellens. Adding Semantic Annotations, Navigation Paths and Tour Guides to Existing Virtual Environments. *Proceedings of the 13th International Conference on Virtual Systems and Multimedia*, pages 50–62. Publ. Springer-Verlag, ISBN 978-0-9775978-1-9, Brisbane, Australia 2007.

[43] M. Kallmann. *Object interaction in real-time virtual environments*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001.

[44] T. Abaci, M. Mortara, G. Patane, M. Spagnulo, F. Vexo, and D. Thalmann. Bridging Geometry and Semantics for Object Manipulation and Grasping. In *Proceedings of Workshop towards Semantic Virtual Environments (SVE 2005) workshop*, 2005.

[45] L. Goncalves, M. Kallmann, and D. Thalmann. Defining behaviors for autonomous agents based on local perception and smart objects. *Computers and Graphics*, 26(6):887–897, 2002.

[46] C. Peters, S. Dobbyn, B. McNamee, and C. O'Sullivan. Smart objects for attentive agents. In *proceedings of The 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 95–98, 2003.

[47] M. Kallmann and D. Thalmann. Modeling behaviors of interactive objects for real-time virtual environments. *Journal of Visual Languages and Computing*, 13(2):177–195, 2002.

[48] B. Pellens. *A Conceptual Modelling Approach for Behaviour in Virtual Environments using a Graphical Notation and Generative Design Patterns*. PhD thesis, Vrije Universiteit Brussel, 2007.

[49] http://ece.uwaterloo.ca/ broehl/behav.html. B. Roehl : Some Thoughts on Behavior in VR Systems, 1995. Last accessed May 28, 2009.

[50] W. Bille. From Knowledge Representation to Virtual Reality Environments. Master's thesis, Vrije Universiteit Brussel, 2002.

[51] http://en.wikipedia.org/wiki/Reflection_(computer_science). Reflection. Last accessed May 28, 2009.

[52] A. Rolling and D. Morris. *Game Architecture and Design: Learn the Best Practices for Game Design and Programming*. Coriolis Group Books, 1999.

[53] http://msdn.microsoft.com/en us/library/ms950721.aspx. XML Serialization in the .NET Framework. Last accessed May 28, 2009.

[54] http://msdn.microsoft.com/en us/library/ms173118.aspx. C# Fields. Last accessed May 28, 2009.

[55] L. Chittaro and R. Ranon. Virtual reality stores for 1-to-1 e-commerce. Technical report, Department of Mathematics and Computer Science, University of Udine, 2000.

[56] G. Haubl and P. Figueroa. Interactive 3d presentations and buyer behaviors. *Conference on Human Factors in Computing Systems Minneapolis, Minnesota, USA*, pages 744–745, 2002.

[57] O. De Troyer, F. Kleinermann, H. Mansouri, B. Pellens, W. Bille, and V. Fomenko. Developing semantic VR-shops for e-Commerce. *Special Issue of Virtual Reality: "Virtual Reality in the e-Society"*, Vol. 11:89–106, 2007.

[58] http://europe.nokia.com/A4195032. Nokia 5300. Last accessed May 28, 2009.

[59] http://www.andytather.co.uk/Panda/directxmax.aspx. Panda DirectX Exporter for 3ds Max. Last accessed May 28, 2009.

[60] http://en.wikipedia.org/wiki/Constraint_satisfaction_problem. Constraint Satisfaction Problem. Last accessed May 28, 2009.

Chapter 8

Appendix

8.1 Smart Entity file format XSD

The XML Schema (XSD) that is displayed below, formally defines our Smart Entity file format.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="SmartEntityDefinition">
 <xs:element name="BehaviorLocator">
  <xs:complexType>
   <xs:attribute name="guid" type="xs:string"/>
   <xs:attribute name="repositoryUri" type="xs:string"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="Variable">
  <xs:complexType>
   <xs:attribute name="name" type="xs:string"/>
   <xs:attribute name="value" type="xs:string"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="SmartEntityDefinition">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="GeometryLocators">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="GeometryLocator">
        <xs:complexType>
         <xs:attribute name="location" type="xs:string"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Semantics">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="StateVariables">
        <xs:complexType>
         <xs:sequence>
          <xs:element ref="Variable" minOccurs="0"
           maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="Members">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Member" minOccurs="0"
           maxOccurs="unbounded">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="MemberBone" minOccurs="0"
              maxOccurs="unbounded">
              <xs:complexType>
               <xs:attribute name="id" type="xs:string"/>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="BehaviorLocators" minOccurs="1"
        maxOccurs="1">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="BehaviorLocator" minOccurs="0"
           maxOccurs="unbounded">
           <xs:complexType>
            <xs:attribute name="guid" type="xs:string"/>
            <xs:attribute name="repositoryUri" type="xs:string"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="Descriptors" minOccurs="1"
        maxOccurs="1">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Descriptor" minOccurs="0"
           maxOccurs="unbounded">
           <xs:complexType>
            <xs:attribute name="visibility" type="xs:string"/>
            <xs:attribute name="name" type="xs:string"/>
            <xs:attribute name="value" type="xs:string"/>
            <xs:attribute name="contenttype" type="xs:string"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="Abilities" minOccurs="1"
        maxOccurs="1">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Ability" minOccurs="0"
           maxOccurs="unbounded">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="Preconditions">
              <xs:complexType>
               <xs:sequence>
                <xs:element ref="Variable" minOccurs="0"
                 maxOccurs="unbounded"/>
               </xs:sequence>
              </xs:complexType>
             </xs:element>
             <xs:element name="Postconditions">
              <xs:complexType>
               <xs:sequence>
                <xs:element ref="Variable" minOccurs="0"
                 maxOccurs="unbounded"/>
               </xs:sequence>
              </xs:complexType>
             </xs:element>
             <xs:element name="BehaviorInvocations" minOccurs="1"
              maxOccurs="1">
              <xs:complexType>
               <xs:sequence>
                <xs:element name="BehaviorInvocation" minOccurs="0"
                 maxOccurs="unbounded">
                 <xs:complexType>
                  <xs:sequence>
                   <xs:element name="Parameters" minOccurs="1"
                    maxOccurs="1">
                    <xs:complexType>
```

```
                     <xs:sequence>
                      <xs:element name="Parameter" minOccurs="0"
                       maxOccurs="unbounded">
                       <xs:complexType>
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="value" type="xs:string"/>
                       </xs:complexType>
                      </xs:element>
                     </xs:sequence>
                    </xs:complexType>
                   </xs:element>
                  </xs:sequence>
                  <xs:attribute name="guid" type="xs:string"/>
                 </xs:complexType>
                </xs:element>
               </xs:sequence>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

8.2 Case study Smart Entity XML

The following XML fragment is the Smart Entity file which was created by using the presented Authoring Tool. The file describes the cellphone which was annotated in the Case Study (chapter 5).

```xml
<?xml version="1.0"?>
<SmartEntityDefinition>
 <GeometryLocators>
  <GeometryLocator location="E:\...\nokia.DAE"/>
  <GeometryLocator location="E:\...\nokia.X"/>
 </GeometryLocators>
 <Semantics>
  <StateVariables>
   <Variable name="open" value="false" />
  </StateVariables>
  <Members/>
  <BehaviorLocators/>
  <Descriptors>
   <Descriptor visibility="PUBLIC" name="Manufacturer"
    value="Nokia" contenttype="TEXT"/>
   <Descriptor visibility="PUBLIC" name="Type"
    value="5200G Black" contenttype="TEXT"/>
   <Descriptor visibility="PUBLIC" name="Camera"
    value="Yes" contenttype="TEXT"/>
   <Descriptor visibility="PUBLIC" name="Weight"
    value="104,2 g" contenttype="TEXT"/>
   <Descriptor visibility="PUBLIC" name="Price"
    value="129,99 euro" contenttype="TEXT"/>
   <Descriptor visibility="PUBLIC" name="Picture1"
    value="E:\...\nokia1.jpg" contenttype="IMAGE"/>
   <Descriptor visibility="PUBLIC" name="Picture2"
    value="E:\...\nokia2.jpg" contenttype="IMAGE"/>
   <Descriptor visibility="PUBLIC" name="Picture3"
    value="E:\...\nokia3.jpg" contenttype="IMAGE"/>
   <Descriptor visibility="PUBLIC" name="Picture4"
    value="E:\...\nokia4.jpg" contenttype="IMAGE"/>
  </Descriptors>
  <Abilities>
   <Ability name="Open">
    <Preconditions>
     <Variable name="open" value="false" />
    </Preconditions>
    <Postconditions>
     <Variable name="open" value="true" />
    </Postconditions>
    <BehaviorInvocations>
```

```
     <BehaviorInvocation guid="TranslationBehavior">
      <Parameters>
       <Parameter name="targetbones" value="top"/>
       <Parameter name="distance" value="150"/>
       <Parameter name="direction" value="0 1 0"/>
       <Parameter name="start" value="0"/>
       <Parameter name="duration" value="2500"/>
      </Parameters>
     </BehaviorInvocation>
    </BehaviorInvocations>
   </Ability>
   <Ability name="Close">
    <Preconditions>
     <Variable name="open" value="true"/>
    </Preconditions>
    <Postconditions>
     <Variable name="open" value="false" />
    </Postconditions>
     <BehaviorInvocations>
     <BehaviorInvocation guid="TranslationBehavior">
      <Parameters>
       <Parameter name="targetbones" value="top"/>
       <Parameter name="distance" value="-150"/>
       <Parameter name="direction" value="0 1 0"/>
       <Parameter name="start" value="0"/>
       <Parameter name="duration" value="2500"/>
      </Parameters>
     </BehaviorInvocation>
    </BehaviorInvocations>
   </Ability>
  </Abilities>
 </Semantics>
</SmartEntityDefinition>
```

8.3 Investigator Camera

This section provides the code that was used for the camera system in our Microsoft XNA based VE applications.

8.3.1 Camera

The Camera class defines an abstract interface for all camera-systems. This allows us to easily change the camera system (at runtime).

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace XNAComponents.Cameras
{
 public abstract class Camera : GameComponent
 {
  public Matrix ViewMatrix;
  public Vector3 TargetVector;
  public Vector3 PositionVector;
  public Vector3 UpVector;

  public Matrix ProjectionMatrix;
  public float ViewAngle;
  public float NearPlane;
  public float FarPlane;
  public float AspectRatio;

  protected Camera(Game hostGame)
        : base(hostGame)
  {
   InitializeMatrices();
  }

  protected virtual void InitializeMatrices()
  {
   PositionVector = new Vector3(0, 2, 15);
   UpVector = Vector3.Up;
   TargetVector = new Vector3(0, 2, 0);
   UpdateViewMatrix();

   ViewAngle = MathHelper.PiOver4;
   AspectRatio = Game.GraphicsDevice.Viewport.AspectRatio;
   NearPlane = 0.01f;
   FarPlane = 200f;
   UpdateProjectionMatrix();
  }
  public abstract void Up();
  public abstract void Down();
  public abstract void Left();
  public abstract void Right();
  public abstract void ZoomIn();
  public abstract void ZoomOut();

  public virtual void UpdateViewMatrix()
  {
   ViewMatrix = Matrix.CreateLookAt(PositionVector, TargetVector,
    UpVector);
  }

  public virtual void UpdateProjectionMatrix()
  {
   ProjectionMatrix = Matrix.CreatePerspectiveFieldOfView(ViewAngle,
    AspectRatio, NearPlane, FarPlane);
  }

  public static Vector3 Unproject(Viewport viewport, Vector3 screenSpace,
   Matrix projection, Matrix view, Matrix world)
  {
   //Convert the VE's 3D coordinates to the screen's 2D coordinates
   //First, convert raw screen coords to unprojectable ones

   Vector3 position = new Vector3();
   position.X = (((screenSpace.X - (float)viewport.X)
    / ((float)viewport.Width)) * 2f) - 1f;
   position.Y = -((((screenSpace.Y - (float)viewport.Y)
    / ((float)viewport.Height)) * 2f) - 1f);
   position.Z = (screenSpace.Z - viewport.MinDepth)

    / (viewport.MaxDepth - viewport.MinDepth);

   //Unproject by transforming the 4d vector by the
   //inverse of the projecttion matrix, followed by
   //the inverse of the view matrix.

   Vector4 us4 = new Vector4(position, 1f);
   Vector4 up4 = Vector4.Transform(us4,
    Matrix.Invert(Matrix.Multiply(
       Matrix.Multiply(world, view), projection)));
   Vector3 up3 = new Vector3(up4.X, up4.Y, up4.Z);
   return up3 / up4.W; //better to do this here to reduce precision loss
  }
 }
}
```

8.3.2 InvestigatorCamera

The InvestigatorCamera allows circular rotation around an object in the VE around the X- and Y-axes. Furthermore, users can zoom in and out. This camera is used in the Authoring Tool.

```
using Microsoft.Xna.Framework;

namespace XNAComponents.Cameras
{
 public class InvestigatorCamera : Camera
 {
  public float speed = MathHelper.ToRadians(1);
  private float zoomspeed = 0.5f;
  private float xRot = 0, yRot = 0;
  private Vector3 translationVector = Vector3.Zero;

  public InvestigatorCamera(Game hostGame)
        : base(hostGame)
  {

  }

  public override void Update(GameTime gameTime)
  {
   Quaternion qRot = Quaternion.Identity;

   if ((PositionVector.Y <= 10) || (xRot < 0))
   {
    if ((PositionVector.Y >= -10) || (xRot > 0))
    {
     Vector3 perpendicularX = Vector3.Cross(
       Vector3.Normalize(PositionVector),
       new Vector3(0, 1, 0));
     qRot = Quaternion.CreateFromAxisAngle(perpendicularX, xRot);
    }
   }

   PositionVector = Vector3.Transform(PositionVector,
    Matrix.CreateRotationY(yRot));
   PositionVector = Vector3.Transform(PositionVector,
    Matrix.CreateFromQuaternion(qRot));
   PositionVector = Vector3.Add(PositionVector,
    translationVector);

   UpdateViewMatrix();
   UpdateProjectionMatrix();

   xRot = 0;
   yRot = 0;
```

```
    translationVector = Vector3.Zero;

    base.Update(gameTime);
   }

   public override void Up()
   {
    xRot += speed;
   }

   public override void Down()
   {
    xRot -= speed;
   }

   public override void Left()
   {
    yRot -= speed;
   }

   public override void Right()
   {
    yRot += speed;
   }

   public override void ZoomIn()
   {
    translationVector = Vector3.Add(translationVector,
      Vector3.Multiply(Vector3.Normalize(PositionVector),
      zoomspeed));
   }

   public override void ZoomOut()
   {
    translationVector = Vector3.Add(translationVector,
      Vector3.Multiply(Vector3.Normalize(PositionVector),
      -zoomspeed));
   }
  }
 }
```

8.4 TemplateBehavior

The TemplateBehavior class is an abstract interface which developers of new behavior implementations for our Smart Entity engine can use.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System.Diagnostics;
using System.Collections.Generic;

namespace TemplateBehavior
{
 public abstract class TemplateBehavior
 {
  private string _guid;
  private Game _hostGame;
  private Model _smartModel;
  private ModelBone[] _targetBones;
  private Matrix[] _originalBoneTransforms;
  private double _duration;
  private double _start;
  private double _position;

  private Matrix _worldMatrix;
  private Matrix _viewMatrix;
  private Matrix _projectionMatrix;

  protected TemplateBehavior(Game hostGame, Model smartModel)
  {
   HostGame = hostGame;
   SmartModel = smartModel;
  }

  public abstract void Initialize();
  public abstract void Update(GameTime gameTime);

  public void UpdateOriginalBoneTransforms()
  {
   if (SmartModel != null)
   {
    OriginalBoneTransforms = new Matrix[SmartModel.Bones.Count];
    //COPY RELATIVE BONE TRANSFORMS!!!!!
    SmartModel.CopyBoneTransformsTo(OriginalBoneTransforms);
   }
  }
```

```
public ModelBone[] TargetBones
{
 get { return _targetBones; }
 set { _targetBones = value; }
}

public double Duration
{
 get { return _duration; }
 set { _duration = value; }
}

public double Start
{
 get { return _start; }
 set { _start = value; }
}

public Game HostGame
{
 get { return _hostGame; }
 set { _hostGame = value; }
}

public Matrix WorldMatrix
{
 get { return _worldMatrix; }
 set { _worldMatrix = value; }
}

public Matrix ViewMatrix
{
 get { return _viewMatrix; }
 set { _viewMatrix = value; }
}

public Matrix ProjectionMatrix
{
 get { return _projectionMatrix; }
 set { _projectionMatrix = value; }
}
```

```
  public Model SmartModel
  {
   get { return _smartModel; }
   set { _smartModel = value; }
  }

  public Matrix[] OriginalBoneTransforms
  {
   get { return _originalBoneTransforms; }
   set { _originalBoneTransforms = value; }
  }

  public double Position
  {
   get { return _position; }
   set { _position = value; }
  }

  public string GUID
  {
    get { return _guid; }
    set { _guid = value; }
  }
 }
}
```

8.5 RotationBehavior

The RotationBehavior class, inherits from the TemplateBehavior and implements a simple linear rotation around a given axis.

```
using System.Diagnostics;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System.Collections.Generic;
using System;

namespace RotationBehavior
{
 public class RotationBehavior : TemplateBehavior.TemplateBehavior
 {
  private float _angle;
  private Vector3 _axis;

 //The constructor should only take arguments that are
 //behavior INdependent and engine/platform dependent!
 public RotationBehavior(Game hostGame, Model smartModel)
         : base(hostGame, smartModel)
 {
  GUID = "RotationBehavior";
  Trace.WriteLine("External Behavior GUID = " + GUID);
  Trace.WriteLine("Invocation on = " + smartModel.Tag);
 }

 public override void Update(GameTime gameTime)
 {
  float absAngle = Math.Abs(Angle);

  //Clamp will yield zero left of the interval
  //and angle right of the interval...
  float angle = MathHelper.Clamp(
   (float)((Position - Start) / (Duration)) * absAngle
   , 0, absAngle);

  Matrix rotationMatrix = Matrix.CreateFromAxisAngle(Axis,
   Math.Sign(Angle) * angle);

  foreach (ModelBone aBone in TargetBones)
  {
   aBone.Transform = rotationMatrix *
    OriginalBoneTransforms[aBone.Index];
  }
 }
 public float Angle
 {
  get { return _angle; }
  set { _angle = value; }
 }

 public Vector3 Axis
 {
  get { return _axis; }
  set { _axis = value; }
 }
}
}
```

8.6 TranslationBehavior

The TranslationBehavior class can perform a linear translation behavior, given a direction (normalized vector) and a distance. The unit of this distance depends on the coordinate system of the 3D model, so it is VE player independent.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;

using System.Diagnostics;

namespace TranslationBehavior
{
 public class TranslationBehavior : TemplateBehavior.TemplateBehavior
 {
  public Vector3 DirectionVector;
  public float DistanceMeasure;

  public TranslationBehavior(Game hostGame, Model model)
           : base(hostGame, model)
  {
  }

  public override void Update(GameTime gameTime)
  {
   //Clamp will yield zero left of the interval
   //and angle right of the interval...
   float factor = MathHelper.Clamp((float)
    ((Position - Start) / (Duration)), 0, 1);
   Vector3 transl = (factor * Distance) * Direction;

   foreach (ModelBone aBone in TargetBones)
   {
    aBone.Transform =  Matrix.CreateTranslation(transl)
     * OriginalBoneTransforms[aBone.Index];
   }
  }
  public Vector3 Direction
  {
   get { return DirectionVector; }
   set { DirectionVector = Vector3.Normalize(value); }
  }

  public float Distance
  {
   get { return DistanceMeasure; }
   set { DistanceMeasure = value; }
  }
 }
}
```