IPT-EGVE Symposium (2007)
B. Fröhlich, R. Blach, and R. van Liere (Editors)

Functional Reactive Virtual Reality

K. J. Blom & S. Beckhaus

interactive media / virtual environments, University of Hamburg, Germany

Abstract

In this paper we introduce a VR system extension that focuses on the creation of interactive, dynamic Virtual Environments. The extension uses the recently developed programming concept, Functional Reactive Programming. This paradigm focuses on an explicit and more natural concept of time in the modeling of dynamics, without sacrificing interactivity. We present an implementation that embeds the Functional Reactive Programming concept into a basic Virtual Reality system, VR Juggler.

Categories and Subject Descriptors (according to ACM CCS): I.6.0 [Computing Methodologies]: Simulation and ModelingGeneral; I.3.7 [Computing Methodologies]: Computer GraphicsThree-Dimensional Graphics and Realism

1. Introduction

Today's Virtual Environments (VEs) are often rather sterile. The single interaction possible in many VEs is moving through the world, which is often also the only dynamic component of the VE. In contrast, the world around us is rather dynamic. Everywhere, something is moving or changing. With a physical environment that is so rich, the typical static VE is only interesting for a short time. Why are VEs not more dynamic and interesting? Modern computer games demonstrate that it is technically possible to have interesting, dynamic environments, engaging players for many hundreds of hours. The thousands of man-hours required to create such an environment is often mentioned as a reason for the difference. Another cited reason is that Virtual Reality (VR) requires interaction in ways that games don't and requires more general solutions, making VR much more difficult to program support structures. One area, where something can be done, is system support for building such dynamic, engaging environments.

The most obvious way of including dynamics in a VE is through standard animation techniques. However, standard animation techniques are not also optimal general solutions for VR. With key-framing techniques, it is difficult to introduce interaction. Inverse kinematics partially overcome the interaction issues with adaptable character movements at run-time; However, it is lacking as a general solution for arbitrary dynamics and for implementing behaviors.

In this paper we present a system based on a new paradigm for the creation of dynamic, interactive environments in VR. Support for creating these environments is achieved using the Functional Reactive Programming (FRP) paradigm [ESYAE94, CNP03]. The FRP paradigm has a small research community laying the foundations of the system, but FRP has not be used in a larger outside project to this point. Here, an implementation integrating FRP into the VR Juggler software system is presented. The use of FRP allows the programmer of the Virtual Environment to describe the dynamic, interactive nature of their environment in a language that more closely matches their understanding of the dynamics, while the underlying Virtual Reality system continues to provide the graphical and hardware interface. The system integration method is specifically chosen to allow the developed FRP-VR concept to be incorporated in various VR systems. A concrete implementation using VR Juggler is given.

The following section presents background on both interactive, dynamic systems in VR and on the Functional Reactive Programming paradigm and how it can be useful in a VR context. Section 3 presents an implementation which couples a recent FRP system with VR Juggler, a system dubbed Functional Reactive Virtual Reality (FRVR). We then discuss the results of our work on FRVR to this point and discuss the directions of our continuing and future work in Section 4. Finally, we conclude the paper.

2. Background

In this section we present background information on two topics. In the first subsection we briefly discuss the support available in current Virtual Reality systems for the creation of dynamic, interactive systems. A cursory overview of the various systems is presented, focusing on the general aspects of the paradigms used. In the second subsection we introduce the programming paradigm, Functional Reactive Programming.

2.1. Support of Interactive, Dynamic VEs in VR

System support for the creation of dynamic and interactive environments in VR is widely varied. Some VR systems provide support only for hardware abstraction. [BJH*01, KBH00] A number of other systems create a dataflow layer for programming dynamics and interactions. On the other end of the spectrum are a few dedicated projects, whose aims are to introduce dynamic and/or interactive components to the world. A survey of all the various systems is too large to be covered in this paper; Instead, we highlight only a few of the more relevant systems.

Various dataflow systems exist and are used by various groups. [BLRS98, Tra99] The largest class of these dataflow systems is the series of systems which are based on SGI's OpenInventor. [Str93] The OpenInventor API is better known to many as VRML. In most of these cases, Scene Graphs (SG) are either designed or retro-fitted to have some sort of overlying layer that forms a dataflow graph. Although it is not inherit to dataflow systems, many of the systems are tightly coupled to the underlying SG and are typically large systems.

The user of these systems uses the dataflow graph to specify the flow of information through the system, typically starting with the system inputs. Dynamics are created, either through the dataflow originating with the system input, the graph's edges shape values to create the dynamics, or the nodes themselves create the dynamics. This final option is performed either through coding in a scripting language or through C/C++ (the more common option). The dataflow itself has no notion of time embedded into it, outside of its frame-locked execution. However, most systems include time as an input to the dataflow system and a few even distribute the time delta since the last frame.

Deligiannidis [Del00] investigated using constraint networks to control dynamics. A network of constraint is used to specify the relationships between components. Deligiannidis's system, DLoVE, had time as explicit component of the design. Additionally, programming is performed with mathematical syntax that is fairly natural. Unfortunately, constraint networks require the programmer to think "backwards" about what they are creating, as one constrains a motion instead of specifying it. The constraint networks approach has number of additional detractors. The most important is that they allow little possibility to have a dynamically changing structure to the network, meaning the world's behavior, as a whole, has to remain constant, i.e. you cannot introduce new objects. DloVE had a limited amount of graph alteration possible, in that one could turn on and off portions of the graph. Constraint networks also tend to have scaling problems, making them currently unsuitable solving for large systems in real-time.

While there has been some interest in developing systems for dynamics, recent research interest seems to be more focused on interaction in Virtual Reality. This body of work has largely investigated how to encapsulate interaction techniques and applying them to objects - almost exclusively static objects - in a general way. The notable exception to this was Zachmann's work, which described a language for describing behaviors and interactions in Virtual Reality. [Zac96] This work was theoretical and the authors are not aware of any work applying this to the actual creation of dynamic, interactive environments.

2.2. Functional Reactive Programming

Functional Reactive Programming (FRP) is a programming paradigm introduced by Conal Elliot. Elliot designed FRP to allow the user to model the animation in, what he felt was, a representation closer to human perception of the motion. [ESYAE94] Elliot's solution was implemented in the pure functional language, Haskell. [Has98] Behaviors are defined via special time dependent continuous functions. The system is capable of reacting to discrete events, by changing which behaviors are active.

As the FRP paradigm is a young principle, it has gone through a number of revisions. Yampa is the current incarnation of the FRP family of languages and the basis for our work. [Cou04, CNP03] While the FRP principle has remained the same, several details have changed in the systems. A new concept in functional programming, Arrows, is used in the implementation to make FRP more flexible and powerful. This has lead to a number of pseudonyms for components, include the secondary name for the system, Arrowized FRP (AFRP), which will be used throughout the paper.

FRP's inherit notion of time is embedded in an implementation of the continuous functions as *signals*, denoted as *Signal Function*s (SFs) in AFRP. SFs in AFRP are implemented as continuations, which allows them to be "frozen" and reactivated. Following the standard pure-functional mantra, SFs require the output of the functions at any time to be dependent only on the input at that time and time itself. However, SFs can be made stateful by the use of a loop, where the output of the function is connected to the input. Furthermore, incorporation of further language extensions make it possible to safely integrate retrieval of outside information, which is important in our implementation.

AFRP provides numerous tools for building dynamic, interactive VEs. The FRP style of programming is based on a building block nature, as is typical of functional programming and provides a number of SF primitives as building blocks. Provided continuous and piece-wise continuous functions include: integral, derivative, hold, and accumulate. Additionally, any standard Haskell function can be made into a SF, allowing the full usage of Haskell's expressive power. The reactive portion of AFRP is based on Events, both external and internal. Numerous functions are also available for the handling of events, specifically with respect to time. Examples are functions that trigger an event after X seconds or at time Y. These create simple and powerful mechanisms for the dynamic VE creator.

The main tool Yampa provides for interactivity and for run-time changes to the complete simulations structure is a series of different event triggered switches. Switches, in their basic form, have a SF that is active until a specified event happens, at which point they switch into another defined function - which can contain any large tree of behaviors underneath it. The various versions of the switch, like the recursive switch, simplify usage, and the special kswitch makes it possible to take a "snapshot" of an SF, capturing its current state. Using this continuation based method, the snapshot can be passed around as another piece of data and reactivated at a later point. This makes a very power tool for the user. Combining the switches with composition SFs, dynamic sets of SFs can be defined and modified at run-time.

Finally, FRP has the advantage of being capable of satisfying the real-time and scalability constraints required for VR. Haskell compiles to quick code, performing almost as well as C++, as evident in the comparison in the Computer Language Shootout [http://shootout.alioth.debian.org/]. This is partially do to Haskell being a "lazy evaluation language," which roughly means that only required values are calculated. Concurrent programming concepts are built into Yampa itself and projects such as Parallel FRP [PTS00] show the potential for expansion, if the simulated environments become too large for simple FRP usage.

3. Functional Reactive VR

In this section we describe a system that has been dubbed FRVR (Functional Reactive Virtual Reality), which integrates a recent FRP implementation into VR systems. This section describes the details necessary for the integration of FRP into VR Juggler and gives an example of its usage.

3.1. Implementation

In our implementation we have elected to use the freely available Yampa (AFRP) libraries for Haskell from Yale University. While AFRP will work with most Haskell compilers, we are currently using the Glasgow Haskell Compiler (GHC), as it provides the highest level of portability and flexibility. The VR portion of the implementation presented here focuses on VR Juggler, as the implementation of FRVR with a basic VR system is slightly less complicated. However, FRVR is designed to be fairly simply integrated in most VR systems and an implementation based in the high-level VR system, AVANGO, has also been written. Both GHC and VR Juggler are cross-platform software systems, providing for a highly portable system.

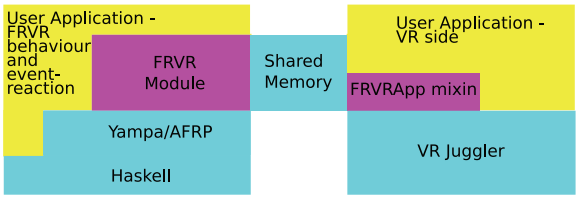

Figure 1: *FRVR's system architecture.*

The FRVR implementation is built using a component architecture. The four basic components are: the bindings to the AFRP/Haskell systems, bindings to the VR system, the connection between the AFRP aspect and VR system, and the FRVR Haskell module. A diagram of FRVR's structure can be seen in Figure 1. In the follow, each of the components is individually handled, starting with the connecting components, and followed by the AFRP/Haskell extensions.

3.1.1. Connecting AFRP/Haskell and the VR system

The connection between AFRP and the VR system consists of two aspects. The first aspect is adapting the control structures of the two systems to get them to work together. This portion deals the setup of the system and is mostly invisible to the user. The second aspect is delivering the input values from the VR system to the AFRP environment and returning the newly calculated values to the VR system, i.e. the run-time aspects.

Yampa's basic design assumes that it controls the main loop, running the simulated environment at the highest rate possible. A Haskell control loop holds a special handle to the AFRP simulation environment and is responsible for the I/O of the inputs and results of the AFRP code. Unfortunately, Haskell's mechanism for referencing such handles from foreign code is currently broken. For this reason, the Haskell environment must be run in a separate thread and synchronized to the VR environment through some other mechanism. This is performed by FRVR using conditionals built into the data passing method described below.

The data passing between the AFRP environment and the VR system could be performed directly through the foreign function interface. However, in order to improve usability, the connection between AFRP and the VR system has been abstracted. Through this abstraction we also help guarantee portability, flexibility, and scalability of the system. The cur-

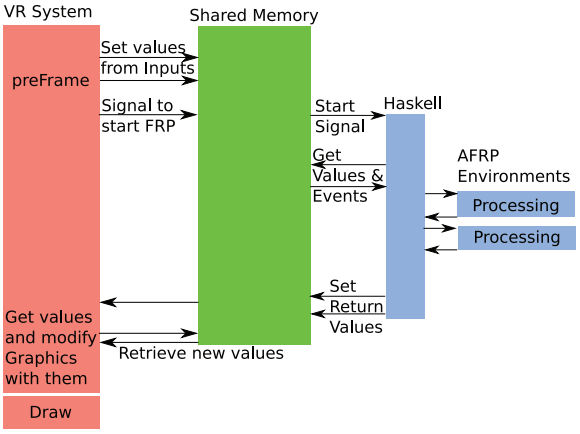

Figure 2: *A diagram of the system flow of the FRVR run-time environment.*

rent version of FRVR uses a simple shared memory space with named tagging of values for the exchange of information between the VR system and the FRP environment. This named tagging allows easier access across the two languages to the information, an idea used in various AI research fields. The memory accesses are controlled through single component with interfaces for both sides. The data structures currently supported by our system are those of the GMTL math library, which VR Juggler uses.

The resultant basic system flow is graphically represented in Figure 2 and proceeds in the following steps:

1. VR system: The inputs to the system are placed in the shared memory
2. VR system: Notification given that the FRP processing should start
3. FRVR: Haskell processing starts
4. FRVR: Haskell retrieves the input values from the shared memory
5. FRVR: FRP simulation environments are called, processing the input events and calculating the new values
6. FRVR: Haskell takes the returned values from the FRP simulation and places them in the shared memory
7. VR system: The processed values are retrieved and packed into the appropriate places in the CG generation.

3.1.2. Interfacing to AFRP

There are several issues to address when interfacing the AFRP built simulation to the outside world. Two of the major issues to be addressed are the handling of data, i.e. how it is inserted into the AFRP environment, and marshalling the data to a format that Haskell can use. The issue of marshalling data, changing it from the external C++ format to a format valid within Haskell, is straightforward. As long as simple types or those composed of a few simple types, Haskell includes functions to handle this. All of this is built into FRVR's Haskell interface to the shared memory, so the user need not worry about it for normal usage.

FRVR currently supports two ways of handling data input. Following the pure functional nature, one needs to have all input delivered into the FRP from the Haskell call, delivered in a special "sense" function. This method has the drawback, that for complex environments, a large data structure must be created to hold all of the input and, then, the individual values have to be distributed to the correct SFs. A second methods involves using a function that works around the IO restriction, breaking pure functionality. By embedding it in a special Arrow, this can be reduced to a single set of code to verify and is how Haskell performs "valid" I/O. FRVR incorporates functions that retrieve and write values from the shared memory, accessed by the tag. This functionality also helps in easing the creation of complex, changing environments, as each behavior can then specify what is required, within itself. The user is free to choose whichever method is appropriate.

3.1.3. Interfacing to the VR system

On the VR system side of the FRVR implementation there are two aspects that have to be handled. The Haskell/AFRP thread has to be started and control loop managed and the data has to be marshalled between the VR system and the shared memory. The system level components are encapsulated into the FRVR system, in this case a mixin-class for VR Juggler's app classes. The shared memory initialization is also handled automatically by FRVR. However, the user must connect the appropriate input to the AFRP system via the shared memory and connect the values generated on the AFRP side to the proper places in their environment manually. This functionality is confined to the main VR application loop: setting inputs, calling the AFRP side, then reading back the generated values.

3.1.4. The FRVR Module

One final aspect of the implementation is the FRVR Haskell module, which is implemented in AFRP and Haskell. The module is composed of the various parts required on the AFRP/Haskell side. A modified version of the AFRP library is included, largely for extensions to the basic control loop. For many dynamics simulations the time delta between loops of the AFRP has to remain small in order to maintain computational stability. A modified control loop allows the AFRP simulation to run with a minimum time delta, regardless of how long the VR loop requires. A related extension allows time skewing.

The other components of the module are: the Haskell shared memory interface, a Math module, and collection of building blocks of common code. Although Haskell is a mathematically based language and has support for many mathematical functions, the standard math of computer graphics, such as vectors, quaternions, and matrices, are not

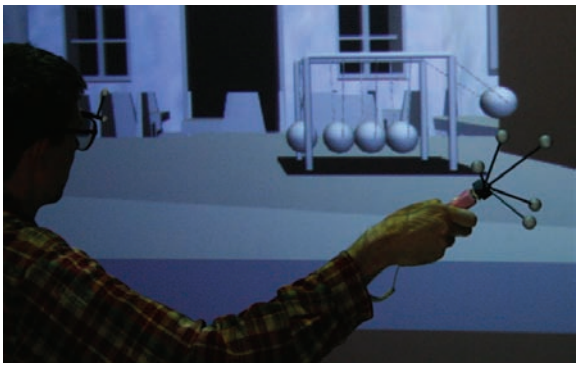

Figure 3: *A user can be seen interating with the Newton's Cradle. When, interacting, the rotation of the wand controls the angle of the ball.*

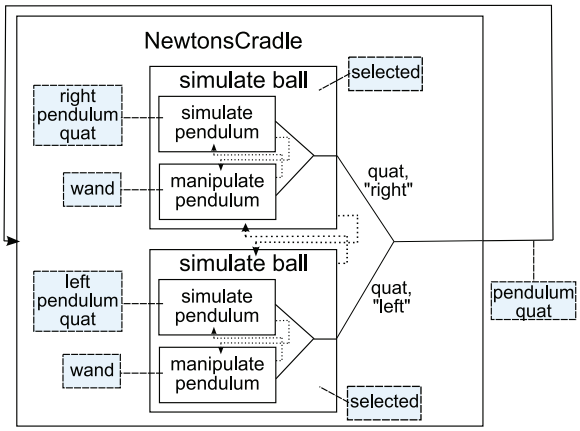

Figure 4: *This diagram illustrates the code structure to handle the dynamics and interaction of the Newton's Cradle. Dotted lines show initialization inputs and the shaded boxes show accesses to the shared memory for data.*

part of the basic libraries. For that reason we have created a library of graphics math functions. The library mirrors the libraries that one finds in many VR systems. The Math types also form the basic types the shared memory system deals with, and values in the memory are automatically marshalled to types in the Math library. Finally, a library of useful functionality for VR environments is also included.

3.2. Example - Newton's Cradle

In this section, we present a small example of the use of FRP for defining dynamic, interactive objects. In this example we create a Newton's Cradle, a popular physics based "office toy" seen in Figure 3.2. While the example is small, it demonstrates how the FRVR system works with FRP to create both dynamics and interaction. It also shows how the pseudo physics of a much more complex system can be simply modeled in FRP. We start with a look at how to code the system's basis, a simple pendulum.

A physical pendulum's motion is defined by many factors. If we describe an undamped, undriven pendulum, the formula reduces to a single equation, yielding the angular acceleration, seen in Equation 1. In the formula L is the length of the pendulum arm. Using a set of differential equations we can retrieve the angle, θ, through a set of integrations.

$$\alpha = -g/L \sin\theta \tag{1}$$

From the mathematical formula one can derive the FRVR FRP code. This function follows the following steps: retrieve the current orientation of the pendulum stored in the shared memory and find the angle it forms with the direction of gravity, the angular acceleration is calculated according to Equation 1, the integral is taken with respect to time to get the angular velocity, and the new rotation is produced by a second integral on the quaternion orientation.

```
testPendulum :: SF () Quatf
testPendulum = proc () -> do
  - -y axis for OGL coords
  ang2grav <- identity -<
      (getQuat "Pendulum") / (Quat 0 0 1 0)
  ang_accel <- arr calcAngAccel -< ang2grav
  ang_vel <- integral -< ang_accel

  -quaternion rotational integration
  rot <- integral -<
         0.5 *^ ((Quat 0  0 ang_vel 0)*rot)
```

On the basis of the pendulum we can create a dynamic, interactive Netwton's Cradle. The diagram in Figure 4 shows the design of the code described here. In the Newton's Cradle, the outer balls behave as a pendulum, when in motion. However, as they make contact with the remaining balls, the momentum of the ball is transfered through the other balls, and the ball on the opposite side begins to travel. This example is interesting, as it is not particularly well suited for a physics engine, due to the many collisions of the balls; However, a pseudo physics can easily be produced with FRVR. This is done using a switch SF and creating an event that occurs when the active ball (SF) collides with the other balls. The transfer of momentum is produced by initializing the a new SF for the opposite ball with the values of the current ball's angular velocity and orientation. This mechanism is configured to be a repeating switch, continuing indefinitely.

The Newton's Cradle example can easily be extended to include interaction. When the user interacts with the ball, the ball simply follows the user's movements. This can be performed in AFRP with a simple SF that retrieves the user's input position or orientation and constrains the derived movement to the pendulum movement. A simple version uses the orientation of the wand, requiring only that the wand's orien-

tation be introduced into the shared memory. When the user releases the ball, simply switching into the system above restarts everything by setting the initial angle to that of the user's interaction. If the user grasps the ball in flight, a switch above the system, waiting on the event triggered from the VR system, can switch once again into the manipulation SF.

4. Discussion and Future Work

During the creation of the FRVR implementation many of the strengths and weaknesses of both the decision to use FRP and its implementation have become apparent. Fortunately, most of the weaknesses have been on the implementation side, and the strengths have been in the design. At this point, we are satisfied with the support that FRVR gives for creating dynamic, interactive environments. The actual implementation of the math, even for complex systems, has been simple, indicating the FRP design is quite effective. The area, for which we feel FRVR could be improved, is in being easier for people to learn and for non-programmers. To that end, a project investigating the use of Visual Programming techniques is currently underway.

Another area of potential future exploration is in relieving the user of the burden of retrieving the information from the shared memory and placing it into the proper place in the graphical system. One possible direction is to couple FRVR more closely with a specific VR system, alleviating or even eliminating the need for programming this connection at such a low level. We have chosen not to do this in our initial work, as it seems an unnecessary constraint on the system's usability and creates an undesirable VR system dependence. Integration in high-level systems already helps alleviate this. Our existing AVANGO bindings, which inserts values into the standard data-flow system, simply the user's work to establishing connections between components.

5. Conclusion

We have presented a new system for VR supporting the creation of interactive, dynamic Virtual Environments. This new system is based on Functional Reactive Programming, a programming concept pioneered by Conal Elliot for computer animation. The FRP implementation, Yampa, is written in Haskell, a programming language with mathematically similar syntax. The FRP adds an explicit time component, allowing the programmer to program dynamics in a continuous manner. Through events, the system is capable of reacting to input, changing the dynamics of the system or even reorganizing the simulation environment at run-time.

The presented system combines the FRP environment with a VR system, creating a system called FRVR (Functional Reactive Virtual Reality). The described implementation incorporates the Yampa system into VR Juggler. The designed system is highly independent from specific VR systems; A second binding to the AVANGO system already exists. We have also shown, via an example creating an interactive Newton's Cradle, how FRVR can be used to create dynamics and interactions with this simulation. In total, FRVR has proven effective for creating interactive, dynamics and forms a system we feel can help advance the field.

References

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proceedings of the Virtual Reality 2001 conference (VR'01)* (2001), p. 89.

[BLRS98] BLACH R., LANDAUER J., RÖSCH A., SIMON A.: A Highly Flexible Virtual Reality System. *Future Generation Computer Systems 14*, 3-4 (1998), 167–178.

[CNP03] COURTNEY A., NILSSON H., PETERSON J.: The Yampa Arcade. In *ACM SIGPLAN Haskell Workshop* (2003), ACM SIGPLAN, pp. 7–18.

[Cou04] COURTNEY A.: *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University, May 2004.

[Del00] DELIGIANNIDIS L.: *DLoVe: A specification paradigm for designing distributed VR applications for single or multiple users*. PhD thesis, Tufts University, Feb. 2000.

[ESYAE94] ELLIOTT C., SCHECHTER G., YEUNG R., ABI-EZZI S.: TBAG: A high level framework for interactive, animated 3D graphics applications. *Computer Graphics 28* (1994), 421–434.

[Has98] HASKELL LANGUAGE AND LIBRARY COMMITEE: *Haskell 98 Language and Libraries The Revised Report*. Tech. rep., 1998.

[KBH00] KESSLER G., BOWMAN D., HODGES L.: The Simple Virtual Environment Library: An Extensible Framework for Building VE Applications. *Presence: Teleoperators and Virtual Environments 9*, 2 (2000), 187–208.

[PTS00] PETERSON J., TRIFONOV V., SERJANTOV A.: Parallel Functional Reactive Programming. In *Proceedings of Practical Aspects of Declarative Programming ({PADL'00})* (Jan. 2000).

[Str93] STRAUSS P. S.: IRIS Inventor, a 3D Graphics Toolkit. In *OOPSLA 93 Conference Proceedings* (Oct. 1993), vol. 28, ACM SIGPLAN, pp. 192–200.

[Tra99] TRAMBEREND H.: Avocado: A distributed virtual reality framework. In *Proceedings of IEEE Virtual Reality* (1999), IEEE Society Press, pp. 14–21.

[Zac96] ZACHMANN G.: A language for describing behavior of and interaction with virtual worlds. In *ACM VRST Conf.* (Hongkong, July 1996).