Supporting the Creation of Dynamic, Interactive Virtual Environments

Kristopher J. Blom[‡]
interactive media / virtual environments
University of Hamburg, Germany

Steffi Beckhaus[§]
interactive media / virtual environments
University of Hamburg, Germany

Abstract

Virtual Reality's expanding adoption makes the creation of more interesting dynamic, interactive environments necessary in order to meet the expectations of users accustomed to modern computer games. In this paper, we present initial explorations of using the recently developed Functional Reactive Programming paradigm to support the creation of such environments. The Functional Reactive Programming paradigm supports these actions by providing tools that match both the user's perception of the dynamics of the world and the underlying hybrid nature of such environments. Continuous functions with explicit time dependencies describe the dynamic behaviors of the environment and discrete event mechanisms provide for modifying the active behaviors of the environment. Initial examples show how this paradigm can be used to control dynamic, interactive Virtual Environments.

CR Categories: I.3.7 [Computing Methodologies]: Computer Graphics—Three-Dimensional Graphics and Realism I.6.0 [Computing Methodologies]: Simulation and Modeling—General;

Keywords: Interactive, Dynamic Virtual Environments; Virtual Reality; Functional Reactive Programming

1 Introduction

The creation of interactive, dynamic Virtual Environments continues to be an often elusive goal of Virtual Reality (VR). Even today, interaction in Virtual Environments (VE) is often restricted to the user's movement through the presented environment. In many cases, this movement through the world is also the only dynamic component of the environment. Modern computer games demonstrate that it is technically possible to have interesting, dynamic environments, engaging players for many hours. The thousands of man-hours employed to create such an environment is one reason for the difference between the quality of VR and game environments in this aspect. Another reason is that VR requires interaction in ways that games do not. VR also requires more general solutions, making it much more difficult to program support structures for VR. One area, where something can be done, is system support for building such dynamic, engaging environments.

The world that is created needs to be more than a static landscape in order to be engaging. Interaction with the world is the classical approach to enhance the environment. The typical VR interactions are simple direct manipulations, like moving objects. A second method of increasing interest is with dynamic components, i.e. things that change over time. This definition of dynamics covers a wide variety of things, instantaneous transitions of a light turning on/off to the kinematics of complex mechanism of a windmill and the behavioral motivations of a complex entity. Finally, interaction with dynamics can be incorporated. Interaction with any dynamics in the world is desirable, not just with human-like avatars.

Creating dynamics, interaction, and their combination is challenging. Interaction, while not simple, is much researched. Dynamics can often be generated using animation techniques. However, this technique is not always optimal. It typically requires a skilled animator for generating the animations. Also, with things such as physical based kinematics, the usual animation methods fall short and hand programming is typically used, even in animation houses. Moreover, interaction with dynamics is very difficult with traditional means. Support for interaction with classical animation methods is limited, again leaving the author to write everything by hand. The best the author can generally hope for is a clock signal to determine how much time has passed. The area for which support has been created in VR is at a level above this, in steering the complete behavior of the environment. While these systems help to specify and control an experience, they ignore the implementation of the actual dynamics and interactivity.

In this paper, we introduce initial work on simplifying the creation process, by using a recently developed programming paradigm, Functional Reactive Programming [Elliott et al. 1994; Courtney et al. 2003]. Functional Reactive Programming (FRP) provides a new approach to incorporate interaction and dynamics together. The FRP paradigm defines the simulation in terms of continuous time behaviors and discrete events effecting the behaviors. FRP works by generating the simulated dynamics of the world itself, but is expanded in our system to include interaction through manipulation of values coming into the simulation from a VR system and through events generated externally. High-level behavioral structures are programmed through switching between running behaviors, based on events occurrences. This model of simulation matches well the structure of interactive, dynamic VEs.

A brief overview of the support available in VR systems is provided in the next section. Section 3 looks at how the FRP paradigm works and shows how our system, which embeds FRP in VR, is built. Initial examples showing how FRP can be used in VR for creating interactive, dynamic VEs are presented in Section 4. Section 5 discusses some of these initial results of using the FRP concept in VR. Finally, we conclude the paper and provide some future directions for research.

2 Support for Interactive Dynamics 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 [Bierbaum et al. 2001; Kessler et al. 2000]. The author is provided with a callback function to write their code, typically in C++. A number of other systems create a data-flow layer for programming dynamics and interactions. On the other end of the spectrum are a few dedicated projects, whose

[‡]e-mail: blom@informatik.uni-hamburg.de
[§]e-mail: steffi.beckhaus@uni-hamburg.de

aims are to introduce dynamic and/or interactive components to the world. Here, we highlight a few of the more relevant paradigms.

Various groups have built systems based on a data-flow concept [Blach et al. 1998; Tramberend 1999]. The largest class of these data-flow systems is the series of systems which are based on SGI's OpenInventor [Strauss 1993], which is better known to many as VRML. In most of these cases, Scene Graphs (SG) are either designed or retro-fitted to have an overlying layer that forms a data-flow graph. Time is typically introduced into the system by inserting a clock time - or occasionally a time delta - into the data-flow each frame. Dynamics are typically created by coding the internal of nodes, either through a scripting language or C++, to create the dynamics based on the frame's time stamp. Interaction with static objects is supported through the data-flow and the same extension of nodes.

Deligiannidis investigated using constraint networks to control dynamics in [Deligiannidis 2000]. 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. The author uses constraints to specify the dynamics of a system, where interaction changes the forces on constraints added for that purpose. DLoVE used a modern constraint system and was able to simulate reasonable sized environments. The DloVE system also introduced a limited amount of graph alteration, allowing one to turn on and off portions of the graph at run-time. This provides some flexibility for the author in terms of dynamically changing world content.

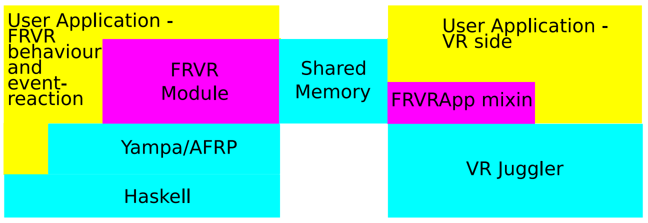
High-level behavioral languages designed for creating interactive dynamic worlds have recently become a focal point in the Web3D community. Various papers over the topic have been presented [Dachselt and Rukzio 2003; Mesing and Hellmich 2006]. The overarching goal of these has been to simplify the creation of interactive dynamics, particularly for non-programming users. In general, the approaches have followed an early theoretical language for describing behaviors and interactions in Virtual Reality from Zachmann [Zachmann 1996]. Both [Dachselt and Rukzio 2003] and [Mesing and Hellmich 2006] propose extensions to the X3D specification to include schemas that describe the high-level dynamics and interactivity. These approaches typically describe the high-level behavior of an environment, without consideration of the actual dynamics implementation.

3 Functional Reactive Programming and its Integration in VR

Functional Reactive Programming (FRP) is a programming paradigm originally introduced by Conal Elliot. Elliot designed FRP to allow the user to model animation in, what he felt was, a representation closer to human perception of motion [Elliott et al. 1994]. *Behaviors* are defined via special time dependent continuous functions. The system is capable of reacting to discrete events, by changing the behaviors that are active.

Yampa is the current incarnation of the FRP family of languages and the basis for our work [Courtney 2004; Courtney et al. 2003]. Yampa is implemented in the pure functional language, Haskell [Haskell Language and Library Committee 1998]. Yampa make use of a new concept in functional programming, Arrows, to improve flexibility and assure that no "space - time" leaks occur.

FRP's inherit notion of time is embedded in an implementation of the continuous functions as *streams*, denoted as *Signal Function*s (SFs) in Yampa. These Signal Functions are implemented as continuations, allowing 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, due to the Arrow based implementation, SFs can be made stateful by using a loop, where the output of the function is connected to the input. In our implementation, we have provided a controlled, encapsulated method for users to break this pure nature for retrieving information from the VR world.


User Application - FRVR behaviour and event-reaction
FRVR Module
Shared Memory
User Application - VR side
FRVRApp mixin
Yampa/AFRP
Haskell
VR Juggler
Figure 1: FRVR's system architecture.

FRP 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. Primitive continuous and piece-wise continuous SFs include: e.g. integral, derivative, hold, and accumulate. Additionally, any standard Haskell function can be made into an SF, allowing the full usage of Haskell's expressive power. The reactive portion of FRP is based on events, both external and internal. Events are modelled as occurrences, i.e. they either exist or do not exist. Throughout the paper, we will use *Event* to indicate the FRP Event type and event when referring to the general concept. Numerous functions are 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.

A series of different event triggered switches are the main tools provides for creating instantaneous interactivity. They can also be used to make run-time changes to the simulation's structure, including changes to the number of simulation elements. Switches, in their basic form, execute a specified SF until a specified event happens, at which point they switch into a secondary SF. The both SFs may contain any set or 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 continuations, a snapshot of the current behavior can be passed around as another piece of data and reactivated at a later point. This makes a very powerful tool for the user. Combined with SFs that compose other SFs in parallel, dynamic sets of SFs can be defined and modified at run-time.

We have developed a component based architecture for incorporating FRP in VR [Blom and Beckhaus 2007]. The structure of the resulting system can be seen in Figure 1; a system we have named FRVR. Yampa is used as a component in the system to implements the dynamics and interactive dynamics of the system. The VR system remains in charge of the main render loop and all of the VR hardware abstraction. In the system presented, VR Juggler is used. Values are exchanged between the two components systems through a special tagged shared memory. The shared memory is implemented as a service accessible from both sides. A second implementation methodology is easily conceivable, coupling the FRP system tightly to the system or SG. In this method, each simulation element would have a call into a FRP simulation. This has not yet be pursued in order to make the implementation portable across VR systems and groups.

4 Creating Interactive, Dynamic VEs

In this section, we present an initial look at implementing interactive, dynamic Virtual Environments with FRP. There are a number of possible approaches to incorporating an FRP simulation in VR. The most basic difference lies in the ownership of the values, i.e. if FRP generates the values itself or if it is only used to incrementally change values from the VR system and return the altered values after calculation. The example given here demonstrates an implementation of autonomous entities in FRVR. FRP controls the entities' dynamics completely. Extending the example, we demonstrate how interaction with such entities can be performed using the FRVR system.

4.1 Boids

Boids are one of the most common place methods for the inclusion of interactive dynamics in VR systems. The idea, introduced by Reynolds [Reynolds 1987], is today used in the gaming community for the implementations of behaviors of various types autonomous entities [Millington 2006]. This "steering behavior" approach, builds the entities' behavior by combining simple rules, where each rule contributes a desired acceleration of the entity. Two of the most common rules, separation and collision avoidance, create an interactive dynamics in the environment. The resultant acceleration parameters are naturally suited to the FRP system, yielding positions and orientations for the entities using the built in integral calculus. The concept of combining various basic functionalities is implemented in FRP, by having each function as its own Signal Function (SF). Each of the basic functionalities is executed, generating the component desired accelerations. These are combined and the integral calculus is evaluated.

A Boid is implemented as an SF itself. The FRP system creates a list of Boid SFs and evaluates them every frame. Below is an excerpt of code that controls a single Boid within the group. Due to the word wrapping here, its appearance is a bit more difficult to read than normal. In general, the special Arrow syntax shows the data flow, from right to left through the arrows. In the first section of the code, steering inputs from the basic steering behaviors are generated. After combining the list of steering inputs, the desired linear acceleration is used to find the new position. Orientation is handled similarly, using quaternion integration.

```
boid :: Boid -> SF (Point3f, [Boid], Point3f) Coord3ff
boid init_boid = proc (target, boid_list, center) -> do
  rec
    cohesive <- group_cohesion -<
      (boid, map (coordPosition.boidCoord) boid_list)
    separate <- separate_group -<
      (boid, map (coordPosition.boidCoord) boid_list)
    dir_travel <- face3D (Quat 0 0 0 1) -<
       (boid, velocity)
    steering_params <- arr accumulate_steering -<
      (weightSteering separate) 0.2) :
      (weightSteering cohesive) 0.2) :
      dir_travel : []

    velocity <- arr (boidVelocity initial_boid+) <<<
       integral -< steeringAccelDirection steering_params
    position <- (transPosFrom initial_boid) ^<<
       integral -<  velocity
```

A library of basis functionalities has been created and differing flocking behaviors can be specified simply by exchanging the active SFs. The more general implementation allows the SFs to be combined using parallel SFs. This enables the Boid's behavioral goals to be altered at run-time. For instance, a higher-level system controlling the priorities for the entity can simply switch in and out

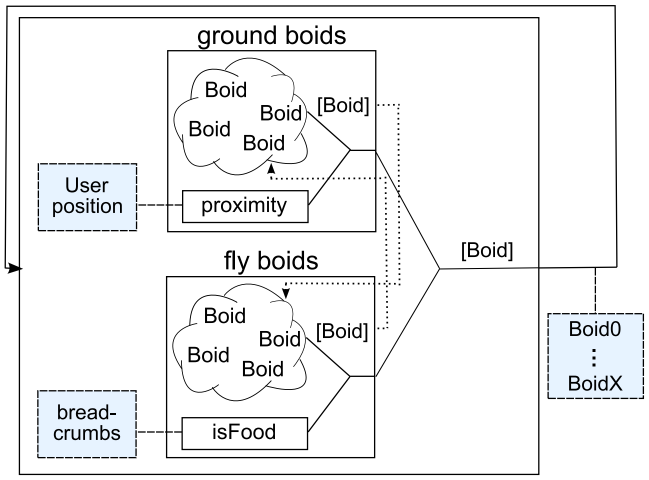


Figure 2: This diagram illustrates the code structure to handle the dynamics and interaction for the user interactive Boids. Dotted lines show initialization inputs and the shaded boxes show accesses to the shared memory for data.

new basis behaviors dependent on which are appropriate at that moment. This differs from the standard approach, where controlling if statements surround sub-behaviors or multipliers are set to zero to eliminate sub-behaviors' effects.

4.2 User Interaction

Adding user interaction can often be simply achieved using FRP's reactive nature. Here, we will investigate how the Boids algorithm can be extended to include interaction with the user space. The interaction is achieved through the user's presence directly, influencing the behavior of the birds. An example of classical VR manipulation can be found in the Newton's Cradle example described in [Blom and Beckhaus 2007].

In [Reynolds 2000] Reynolds describes a demonstration program for a game console, "Pigeons in the Park," where the user could steer an RC car into a group of pigeons feeding on the ground to make them take off and fly around. This high level behavior change was implemented using a Finite State Machine in Reynolds system, where each state's behavior was defined by a different set of basis behaviors. Here, a modified version of the example, using the user's presence in the world, can be implemented using FRVR.

To program this is in FRP, we first note that the Boids now have two different sets of behaviors, feeding on the ground and flying. Assuming that both these behaviors can be described using variations and combinations of the standard steering algorithms, two independent behaviors are created, one for flying and one for the ground. The program of each behavior follows roughly that of Section 4.1.

With the two behaviors programmed, we then need the conditions that define which behavior is currently active. In this case, the approaching user causes the birds to fly away. An event, based on user proximity, can be used. As birds have a tendency to scare as a group and all fly away, a single bird taking flight, can be used for making the whole group fly. The reason for the birds return to the ground is not obvious, but we will assume that a feeding condition can be triggered by the user, for instance by placing some bread on the ground. In this case, the user interaction with a secondary object causes the event that triggers the behavior change.

Figure 2 provides a graphical representation of how the program is structured for such a Boids implementation. Both of the Boids behaviors are shown inside of a switch. When the ground Boids are active, a special SF, inside of each Boid, detects the proximity of the user, retrieving the user's position from the shared memory. If a fly event is generated by one individual Boid, the switch will switch into the *fly* behavior. In the description above, we ignored that the Boids have to remain consistent, requiring that the values for the Boids, e.g. position in the world, have to be handed over for initialization of the other behavior. The *fly Boids* reacts to a single external event asserted by the VR side, breadcrumbs, as a group. This causes a switch back into the *ground Boids* behavior. The transitions between methods have to be handled specially. This can be achieved by writing the appropriate functionalities and switching into them between behaviors.

5 Results and Discussion

In order to test the performance of FRP, we have compared the FRVR implementation of Boids with that of a traditional C++ implementation. The programs were run with a basic set of 1000 Boids, a number that created a significant system load on a 2.1 Ghz P4. In this test, the FRVR system performed 30% better than the C++ implementation. While we had expected comparable performance from FRP, an improvement of this size with the overhead of the shared memory exchange was unanticipated. We suspect that this difference is due to two factors. The compiler optimized Haskell code the FRP code performs exceeding well. 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. The C++ implementation was coded using Object Oriented principles that may have played a significant role in the efficiency of the code, particularly due to virtual function calls. In general, we expect speeds of FRVR code to approximate those of native low-level languages.

6 Conclusion

In this paper we have shown preliminary examples of the creation of dynamic, interactive Virtual Environments, using the recently developed paradigm Functional Reactive Programming. By embedding FRP in a VR system, such as VR Juggler or AVANGO, the programmer of the environments behaviors can approach the VE as a hybrid system of continuous functions and discrete events. The continuous functions define the time dependant aspects and the discrete events are events to which the system reacts. The developed Functional Reactive Virtual Reality system delivers the speeds necessary for VR systems, matching C++ implementations of behaviors, while providing the continuous reactive programming offered by FRP. We feel the combined system presents the potential to ease the development of interactive, dynamic environments and move VR in the direction of creating true interactive experiences.

Our continuing work with FRVR will further explore the potentials of the FRP paradigm for controlling dynamic, interactive environments. The FRP paradigm is well suited to higher-level concepts, such as interactive storytelling. We are currently working on a building a more complex world to explore FRVR's usability. Another direction of interest is work on making the author's work easier. To this end, we are working on developing a graphical interface for FRP coding. Finally, using FRVR as an implementation language for some of the high level behavior schemas developed in VR previously would be advantageous, as it provides tools at both the low level dynamic level and system wide behavioral level.

References

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

BLACH, R., LANDAUER, J., RÖSCH, A., AND SIMON, A. 1998. A Highly Flexible Virtual Reality System. *Future Generation Computer Systems 14*, 3-4, 167–178.

BLOM, K. J., AND BECKHAUS, S. 2007. Functional Reactive Virtual Reality. In *Short Paper Proceedings of the IPT/EuroGraphics workshop on Virual Environments (IPT-EGVE '07)*, EuroGraphics.

COURTNEY, A., NILSSON, H., AND PETERSON, J. 2003. The Yampa Arcade. In *ACM SIGPLAN Haskell Workshop*, ACM SIGPLAN, 7–18.

COURTNEY, A. 2004. *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University.

DACHSELT, R., AND RUKZIO, E. 2003. Behavior3d: An XML-Based Framework for 3D Graphics Behavior. In *Proceedings of the ACM Web3D 2003 Conference*, ACM Press.

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

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

HASKELL LANGUAGE AND LIBRARY COMMITTEE. 1998. Haskell 98 Language and Libraries. Tech. rep.

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

MESING, B., AND HELLMICH, C. 2006. Using Aspect Oriented Methods to Add Behaviour to X3D Documents. In *Proceedings of the ACM Web3D 2006 Conference*, ACM.

MILLINGTON, I. 2006. *Artificial Intelligence for Games*. Morgan Kaufmann.

REYNOLDS, C. W. 1987. Flocks, herds, and schools: A distributed behavioral model, in computer graphics. In *SIGGRAPH '87 Conference Proceedings*, vol. 21, 25–34.

REYNOLDS, C. 2000. Interaction with groups of autonomous characters. In *Game Developers Conference 2000*.

STRAUSS, P. S. 1993. IRIS Inventor, a 3D Graphics Toolkit. In *OOPSLA 93 Conference Proceedings*, vol. 28, ACM SIGPLAN, 192–200.

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

ZACHMANN, G. 1996. A language for describing behavior of and interaction with virtual worlds. In *ACM VRST Conf.*