ENGINEERING A SOFTWARE SYSTEM OF REUSABLE TRAVEL PRIMITIVES FOR VIRTUAL ENVIRONMENTS

Kevin Meinert, Laura Arns, Carolina Cruz-Neira
Virtual Reality Applications Center, Iowa State University

ABSTRACT

Travel techniques are used to control a user's viewpoint in virtual environments (including simulation, training, and first person games). Effective travel is critical, because travel is present in nearly every virtual environment application. Previous research on virtual travel focused on the theory and classification, but few implementation tools have been developed. Our work focuses on a practical approach to implement travel methods. We assert that the large number of existing travel methods all share common "primitive elements". Examples of primitive elements include: gravity, collision response, friction, and spring force. Primitives like these can be extracted, then mixed and matched to create many different travel methods. Ad hoc ways of implementing travel methods lead to designs that are single purposed and brittle. By separating the primitive elements of travel, we can use them to construct many navigation metaphors. This modular approach leads to a solution that can be extended and refactored easily. With travel primitives, the system designer can tweak the overall navigation metaphor until it is optimal for the application user. The system should also be flexible enough to accommodate future travel primitives. This promotes longevity and adoption of the toolset by a wide user base. Most previous travel methods have been implemented for particular applications and not for general use. This paper presents the design and development of a toolset of travel primitives for general use by VR developers. We also explain how our own classification system for virtual travel led to the desire to create this toolset. The toolset is structured into 3 layers: Dynamic Animation System: Provides an extensible animation system for particle or rigid body dynamics, allowing addition of arbitrary operations and objects. Travel Primitives: Provides primitive elements of travel with operators for the animation system. Travel Methods: Composes primitives into a navigation metaphor such as drive, walk, or fly. We also discuss the implementation of this design to date, and several case studies.

ABOUT THE AUTHORS

Kevin Meinert is a graduate research assistant at the Virtual Reality Applications Center at Iowa State University. He is currently pursuing an M.S. in Computer Engineering from Iowa State University. He also holds a B.S. in Computer Engineering from Iowa State University (Dec. 1999). Mr. Meinert's research interests are in animation and physics, sound synthesis, and content creation in virtual environments. He has been a member of the VR Juggler development team since 1998. He also runs the virtual reality news web site http://www.vrsource.org.

Dr. Laura Arns received her Ph.D. in Computer Science from Iowa State University in May 2002. She also holds an M.S. (1998) in Computer Science from Iowa State University and a BA (1996) in Computer Science and Mathematics from Wartburg College. Dr. Arns was the recipient of the 11th Annual I/ITSEC Graduate Student Scholarship. She is currently a postdoctoral researcher with the Virtual Reality Applications Center, where she has worked on a number of projects ranging from multivariate statistical data visualization to oil field exploration. Her research interests are in the areas of applied virtual environments, human factors in virtual reality, and virtual reality usability.

Dr. Carolina Cruz-Neira is an Associate Professor in the departments of Industrial and Manufacturing Systems Engineering and Electrical and Computer Engineering at Iowa State University (ISU). She is also the Associate Director of the Virtual Reality Applications Center. Her research focuses on complex software systems, usability studies and the application of virtual reality in the Sciences, Engineering, Art and Humanities. Dr. Cruz has a Ph.D (1995) in Electrical Engineering and Computer Science (EECS) from the Electronic Visualization Laboratory(EVL) at the University of Illinois at Chicago. Her Ph.D. dissertation included the design of the CAVE ™ Virtual Reality Environment, and the CAVE Library software specifications and implementation. She has an MS (1991) in EECS from EVL at the University of Illinois at Chicago and a BS (1987- Cum Laude) in Systems Engineering at the Universidad Metropolitana of Venezuela

.ENGINEERING A SOFTWARE SYSTEM OF REUSABLE TRAVEL PRIMITIVES FOR VIRTUAL ENVIRONMENTS

Kevin Meinert, Laura Arns, Carolina Cruz-Neira
Virtual Reality Applications Center, Iowa State University

INTRODUCTION

Travel techniques, or locomotion, are used to control a user's viewpoint in virtual environments [Bowman98] (including simulation, training, and first person games). Nearly every virtual environment (VE) makes use of some form of travel. It is extremely important for the user to be able to navigate successfully in the virtual world in order to achieve his goals in the VE. Therefore, effective travel techniques are critical.

In VEs we need many of the same travel methods as in real life, and we can also use methods that cannot exist in the physical world. Today many VE applications implement their travel methods from scratch , ignoring prior tools and previous travel implementations. Because there are so many travel method variations available, and an increasing number of VE applications, a tool is needed to facilitate arbitrary travel method design.

Previous research on virtual travel focused on the theory and classification, but few implementation tools have been developed. Our work focuses on an extensible toolkit to enable modular implementation of travel methods. We assert that the large number of existing travel methods all share common "primitive elements". Examples of primitive elements include: gravity, collision response, friction, and steering force. These primitives can be extracted, then mixed and matched to create many different navigation methods.

BACKGROUND

Travel allows users to gain access to areas or regions of their environment otherwise not available to them. This is also true in real life. We navigate every day using travel methods such as walking, driving, flying, clicking, dragging, and typing. In virtual environments we need many of the same travel methods as we do in real life, along with additional methods that cannot exist in the physical world, but which allow easier access.

Here are some classic examples of travel methods:

i. Drive or walk somewhere in order to experience, examine, or otherwise engage that destination in some action.

ii. Teleport to some location for convenience, so that time is not wasted on travel.

iii. Fly up and around architectural models or other multidimensional data to view at convenient angles.

iv. Zoom in to do fine detailed editing and then zoom out to examine the work as a whole.

These and many more travel methods are possible, but time, effort, and expertise are required to make them available to the user.

Theoretical research has identified a large number of navigation metaphors such as World In Miniature [Stoakley95], Scene In Hand and Eye in Hand [Ware90], Leaning [Fairchild93], and many others. Unfortunately, metaphors such as these are usually developed in isolation. Different travel metaphors may have little or nothing in common, making transitions between different VE applications confusing for users. In addition, comparisons between metaphors are difficult for VE developers because it may not immediately be apparent what advantages and limitations each metaphor provides, and what features different travel methods may share. These difficulties result in VE applications being developed in an ad-hoc fashion, with developers implementing various travel methods in a somewhat random fashion until they discover one that works well for the new application. Such an approach can waste large amounts of time and money, and also fails to take advantage of common features which all travel methods utilize.

A new approach is to create taxonomies, which classify travel methods according to various components, such as the method for the user to indicate a direction of travel. One such taxonomy has been developed by [Bowman98]. Such a taxonomy allows developers to identify the features which different travel metaphors share, and where they differ. Another taxonomy was developed by some of the authors of this work, which includes elements such as display devices and VE interaction devices [Arns02].

Although much research has been done on the theory and classification of navigation in virtual worlds, general use tools for implementing those theories have seldom been developed. Our work focuses on creating a practical approach to implement travel methods.

Although there are many travel methods, they all share a set of common "primitive elements" that compose each method. The entire set of travel primitives is bounded by the creativity of the programmer, but includes animation and control primitives. Some examples of these primitives are gravity, collision detection/response, friction, spring force, interpolation along a curve or line, and teleportation. These primitives can be mixed and matched to create many different navigation methods. The resulting navigation methods can be subtly or completely different from each other depending on the needs of the application or creativity of the author. For example, to achieve a method we could call "walking with a head-bob", mix together gravity, keyframing, collision detection/response, and acceleration. Given a list of primitives, each could be used to enhance an existing prepackaged travel method. For example, when using our head-bob walk travel method, the addition of a jump primitive could allow the user to access parts of the virtual environment otherwise inaccessible.

In addition to serving as building blocks, the travel primitive approach provide the benefits of well-engineered software: reusability, predictability, portability, and extensibility. Reuse provides the designer with a set of well-tested prefabricated items that they can use to quickly assemble their design without reinvention. Predictability in expected results is valuable to rapid prototyping of a design because it gives confidence that the system will perform as specified. Portability between input devices, application types, and computing systems is also very important to the reuse of the software and to the flexibility of the application. Low portability of the toolkit directly impacts the portability (and thus reuse) of the application. Extensibility is important so that a designer can simply "plug and play" with different travel primitives and even add their own custom primitives. This allows the designer to tweak the overall travel metaphor until it is optimal for the application user. It is important that the system be flexible enough to accommodate new travel primitives to promote the longevity and adoption of the toolset by a wide user base. Ad hoc ways of implementing travel methods lead to designs that are single purposed and brittle. Separation of primitive elements of travel and their use in the construction of the travel metaphor leads to a design that can be extended, refactored, and shared with others easily.

The system we present in this paper aims to minimize these obstacles. We offer a framework allowing plug and play building of methods through aggregation of prebuilt travel primitives. The framework has a common application programming interface (API) making it possible to share prebuilt method components with other applications. Finally, the toolkit is a place where prebuilt methods can be submitted to be shared with other applications, so that future projects can simply reference any travel component as a "black box". This componentization of travel methods enables rapid prototyping as well as opens a niche to component developers.

The Travel Pattern

Our travel system software represents a flexible implementation of a recurring design pattern [Gamma95] in VEs that we call the "Travel" design pattern [Meinert02]. The Travel Pattern is a software design pattern for virtual environments that details how to build a travel method. More specifically the Travel Pattern is a cookbook recipe of the software components of a travel system. A travel method is a way to move a user s viewpoint such as "drive", "fly", "swim", "walk with a head bob", "drive with 4 wheel suspension", etc. The Travel pattern involves a camera, avatar, travel primitives, collision, input, and rendering:

- **Avatar**. An avatar is used to know where the user is located in the virtual world, for collision detection and rendering. The avatar is generally controlled by user input, and contains at least a data representation of the object (often the user) that is traveling. Travel in VR always moves an avatar, and the avatar is sometimes partially or fully rendered in the VE.
- **Camera**. A camera is used to control a user s viewport. In some travel methods, the camera is placed at the same location as the avatar. The camera represents a single transform in 3D world space coordinates, and is not usually rendered. The camera is a concept that describes the application user's view into the world. Without the camera the VE could not be seen.
- **Auxiliary Objects**. In some travel methods (usually for head tracked setups), a parent child relationship is needed. For example, in a CAVE“ [Cruz93] system the user moves their avatar within a navigatable "platform . In some complex travel methods, there is a head attached to the avatar, or other linkages like arms. Vehicle simulations have wheels attached to the avatar.

- **Travel Primitives**. Primitives such as gravity, jump, steering force, and acceleration can be composed together to form the travel method. The primitives hold the logic for controlling the avatar, camera, and/or platform objects. Primitives can be decoupled so they can be interchanged and reused in other travel methods.

- **Animation Controller**. An animation controller is a controller that can influence transforms over time where the end result of these transforms may or may not be used for rendering. An animation controller of some form is needed for the viewpoint motion control. To move the avatar and camera around the VE space, we need some way to update their transforms. The fact that viewpoint motion control is needed suggests we need some system to manage this motion. In some travel systems the animation controller is tightly coupled together with the other attributes described in this pattern, but doesn't need to be as we will see in the section called Gator . Travel systems like Gator use a rigid body dynamics system for very generalized control of animation. Other methods like key framing or kinematics may also be used depending on the travel system.

- **Collision System**. A collision system is necessary so that the travel method knows about its environment, and can react to it. Given a set of geometry and an avatar (or some object), a collision system returns collision information [Eberly01]. Collision detection by itself is a broad topic, ranging from navigation to computer games to robotics. More information on this is available in computational geometry, graphics, and game references. The travel primitives needing information about the scene geometry use the collision detection system.

- **Input System**. An input system allows the user to control the travel method, and gives data useful for controlling the travel system. This system is an abstraction on top of user-input hardware, and sometimes software GUI widgets or other control signal sources. An input system is used by various input operators that are plugged into the animation system. Upon receiving input, these operators can affect the avatar and parameters of the other operators in the animation system.

- **Rendering System**. Some way to render the avatar and/or camera transforms is needed. See the section called Using Gator in Applications for more discussion on rendering a travel method.

GATOR

The software we present here is named Gator (as in Navi gator). Gator is an open source C++ toolkit [Gator02] that allows arbitrary composition of travel primitives into travel methods. Gator provides an object-oriented software implementation of the Travel Pattern described in the previous section. The benefit of Gator over previous Travel implementations lies in its flexibility, allowing users to specify new travel methods at a finely grained level.

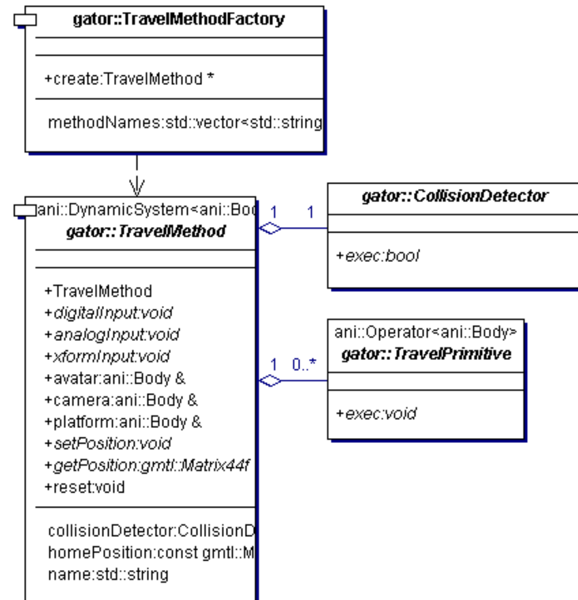

Figure 1. Gator Software Design

Gator defines a framework to create travel methods (Figure 1). Each travel method created with Gator has the ability to be shared with other applications generically as "black boxes". It implements the Travel Pattern in a general way, offering pluggable travel primitives, generalized input, and an application definable collision system.

A travel primitive is a small interchangeable element that creates a travel method (like gravity, steering force, collision response, accelerate). Internally, Gator uses a rigid body dynamics engine so that any physical phenomenon can be modeled. In addition to this method of viewpoint animation [Moller99], other forms of animation may be used as well.

It comes with several predefined travel primitives discussed later in the Case Study section. When building travel methods, users can create their own custom travel primitives or use the ones that come with Gator.

Gator generalizes input so that each travel method may be used generically by any application. We believe that the application doesn't need to know intimate details of the travel method. The generalized input API helps to keep the travel method a black box to the application.

The collision detection system in gator is abstracted so users can define their own collision methods if needed. Gator cannot possibly include support for every collision system, scenegraph, or other variation, so we allow the application to implement a collision adaptor if necessary. It also provides a few default collision detectors for convenience, and specialized abilities can be added later when needed.

Implementation

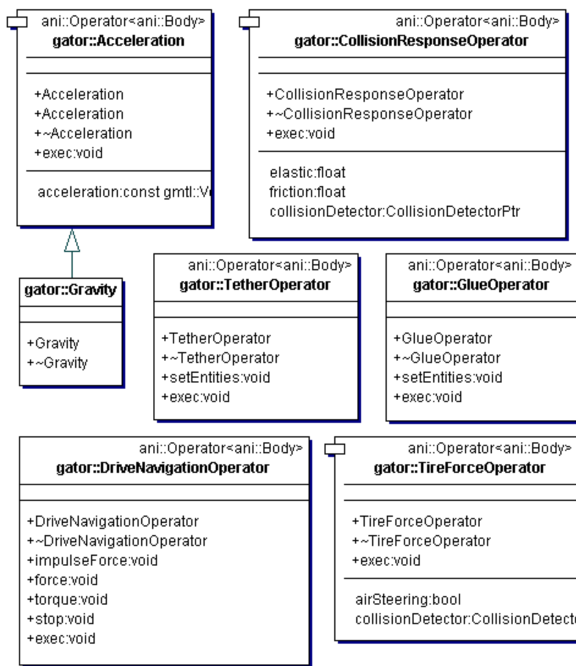

Figure 2. Example travel primitives included with Gator, each is a pluggable operator. Listed are: Accelaration, CollisionResponseOperator, Gravity, TetherOperator, GlueOperator, DriveNavigationOperator, TireForceOperator.

The core of Gator is a rigid body dynamics system that allows dynamically plugable operators and objects [Baraff98] for generality. An operator is a Strategy [Gamma95] that performs a specific action (some examples: apply force of gravity, set position to input device). Each travel primitive is implemented with a pluggable operator (Figure 2). The operators can be composed to form a travel method. A dynamics system was chosen because it allows many types of animation, not just physically based animation. Several animation methods are supported by this system ranging from physics to key frame and other animation methods. Gator also allows developers to add their own travel primitives and therefore increase the flexibility to build diverse travel methods. To allow Gator to be portable to several subsystems, Adapters [Gamma95] can be written for collision, input and rendering.

Using Gator in Applications

To use Gator in an application is fairly simple. Gator provides a `TravelMethod` Factory [Gamma95] that allows the application to request a prefabricated travel method from a list of known methods. Otherwise the application is free to define its own travel methods, or extend existing ones.

The application will need to connect the travel method for rendering, collision, and input (explained in the following paragraphs). Rendering, and often collision, is dependent on the application's graphics system.

For rendering in OpenGL, the transform returned by `TravelMethod::camera()` should be transferred on each frame using the OpenGL function `glMultMatrixf`. For collision detection, it is up to the user to supply a `CollisionDetector` function object so the `TravelMethod` can respond to the scene correctly.

To hook up the `TravelMethod` when using a scenegraph, a `TravelMethodNode` can be added to the scenegraph which would aggregate the actual `TravelMethod` and copy over the `TravelMethod::camera()` transform in the node's application callback. A `CollisionDetector` function object can be used to intersect directly with the scenegraph root node, an external collision data structure, or some other method needed by the application.

Depending on the travel method, the application may want to render the avatar or parts of the avatar. `TravelMethod::avatar()` can be used to place the avatar geometry in the scene. Each `TravelMethod` has input terminals accessible through `TravelMethod::digitalInput`, `TravelMethod::analogInput`, and `TravelMethod::xformInput`. The application needs to call these with their input device states on each frame. Finally, the application needs to call `TravelMethod::step()` with the time change since the last frame.

Example of Gator in Use

While Gator is a relatively new system, there are now a few applications using Gator as a travel component in virtual environments.

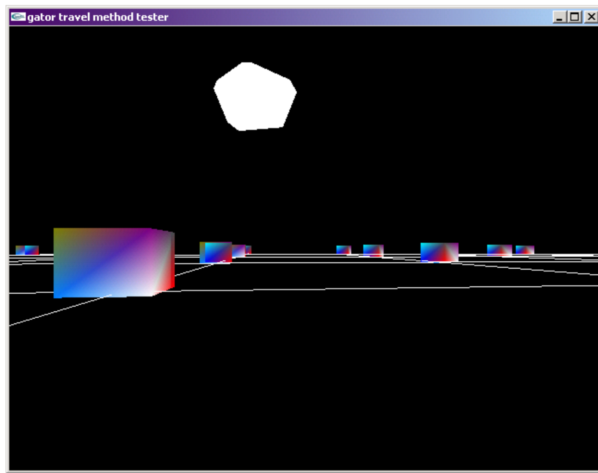

Figure 3. Gator-Nav, a sandbox test application for new travel methods. Shown is a 3rd person travel method where the avatar (depicted as a white polyhedron) is jumping.

Gator-Nav. Included in the Gator source tree is a sandbox test application (Figure 3) for `TravelMethods`. The application uses the `TravelMethodFactory` to create any registered method requested by the user. This application allows the user to select a method and try it out in the virtual environment.

Case Studies

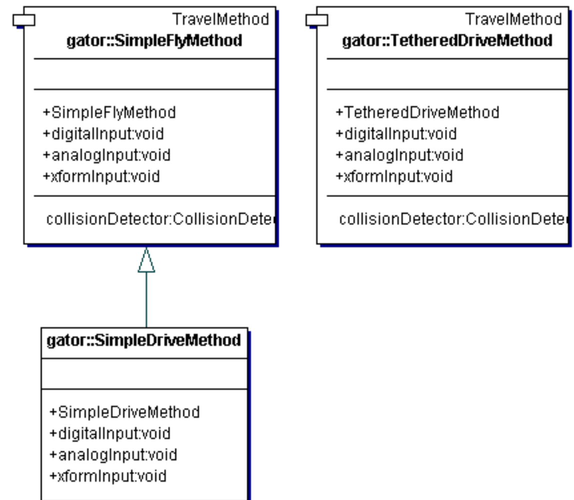

Figure 4. Gator includes some predefined travel methods. (Notice that SimpleDriveMethod extends SimpleFlyMethod, shown by the UML language arrow notation). Also shown here is TetheredDriveMethod.

Gator includes some predefined travel methods (Figure 4). Constructing these methods involved adding some common primitives (Figure 2). For example, they all reuse a collision/response primitive, and an automobile-style accelerate/decelerate model. Next are the details for how each method was built.

SimpleFlyMethod

This is a travel method that implements the common "fly" metaphor found in many VR systems. It is composed of the following primitives:

- Automobile-style accelerate/decelerate. This primitive applies force to the avatar in the direction it faces. Reverse and stop forces are also triggerable.
- Steering Force. This primitive applies force to keep the avatar from sliding sideways as it turns. It simulates the force we experience from the tires on a car while turning. While not realistic, this primitive is very useful to keep the navigation under control.
- Collision Response. This primitive does collision check on the scene geometry, and feeds back the appropriate force to the avatar.
- Glue. This primitive attaches the camera to a place near the avatar s head. It causes the camera s transform to equal the avatars with some offset.

SimpleDriveMethod

This is a travel method that implements the common "drive" metaphor found in many VR systems. It is composed of the following travel primitives:

- This travel method extends the `SimpleFlyMethod` adding a Gravity primitive. The Gravity primitive applies a constant acceleration downwards to the avatar.
- A flag is set in the Steering Force primitive that tells it to steer only when colliding. This allows the realistic effect that while in the air , the avatar can slide sideways. This design choice may not be the easiest to control, so an application designer may choose to disable the condition and leave the primitive on at all times.

Tethered Drive Method

This travel method is similar to Mario64 style control [Mario64]. Mario64 is an entertainment oriented VE application found on the Nintendo 64 game console. This style of navigation may be interesting in a VR environment as well. Here are the primitive elements:

- Automobile-style accelerate/decelerate. See SimpleFlyMethod for description.
- Tether between avatar and camera. This primitive keeps the camera following behind the avatar for a 3rd person effect. This personal camera crew for the avatar is written so that it tries to stay between 6 and 12 meters behind the avatar. The tether is not a hard constraint, and allows the camera to drift a little.
- Steering Force. This primitive applies force to keep the avatar from sliding sideways as it turns. It simulates the force we experience from the tires on a car while turning.
- Collision Response. This primitive does collision check on the scene geometry, and feeds back the appropriate force to the avatar and also to the camera.
- Gravity. This primitive applies a constant downwards acceleration to the avatar and camera.

CONCLUSIONS

One of the biggest problems faced today is a lack of good open source general use tools for travel in VEs. For example, a biologist should not have to focus much time on defining and implementing a travel method for their VE, they should be able to focus wholly on the biology problem at hand.

Creation of travel methods for virtual environments can consume time and requires domain expertise to implement well. The system we have presented in this paper aims to minimize the obstacles found when implementing travel methods.

The Gator toolkit allows plug and play building of methods through aggregation of prebuilt travel primitives. The framework has a common interface making it possible to share prebuilt method components with other applications.

Finally the toolkit is a place where prebuilt methods can be submitted to be shared with other applications, so that future projects can simply reference any travel component as a "black box". This componentization of travel methods enables rapid prototyping and opens a niche to component developers for the sharing of travel methods between interested groups.

FUTURE WORK

Currently Gator includes the engineering components necessary for implementing general-purpose virtual travel, but lacks some of the user-centered interface features required by the taxonomy in [Arns02]. To support this taxonomy, the Gator API needs to be extended to support more flexible input schemes. With these schemes in place, sample modules that implement these techniques could be created.

Authoring travel methods is currently possible in Gator through C++ programming, but could be made easier through scripting or a visual interface. Gator is able to serve as the subsystem for user-level visual travel method authoring tools, which will be needed for consumer-level VE authoring. These authoring tools would be an interesting point of future research.

ACKNOWLEDGEMENTS

This work was supported in part by the 11th Annual I/ITSEC Scholarship, and Iowa State University.

REFERENCES

[Arns02] L. Arns, C. Cruz-Neira, A New Taxonomy for Locomotion in Virtual Environments. Proceedings of 4th Virtual Reality International Conference, Laval, France, June 2002, pp. 141-149.

[Baraff98] D. Baraff. A. Witkin. Physically Based Modeling . Association for Computing Machinery (ACM) Siggraph 98 Course Notes. 1998.

[Bowman98] D.A. Bowman. D. Koller. L.F. Hodges. A Methodology for the Evaluation of Travel Techniques for Immersive Virtual Environments . Virtual Reality: Research, Development, and Applications. 3. 2. 1998. 120-131.

[Bowman99] D.A. Bowman. D. Johnson. L. Hodges. Testbed Evaluation of VE Interaction Techniques . Proceedings of ACM VRST. 1999. 26-33.

[Cruz93] C. Cruz-Neira, D.J. Sandin, T.A. DeFanti, Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. Proceedings ACM SIGGRAPH, p. 135-142, 1993.

[Darken96] R. Darken96. J. Sibert. Wayfinding Strategies and Behaviors in Large Virtual Worlds . ACM CHI. 1996. 142-149.

[Eberly01] D. Eberly. Collision Detection ch.6 . 3D Game Engine Design. Morgan Kaufmann Publishers. 2001. Academic Press.

[Fairchild93] K.M. Fairchild. B.H. Lee. J. Loo. H. Hg. L. Serra. The Heaven and Earth Virtual Reality. Designing Applications for Novice Users . VRAIS. 1993. 47-53.

[Gamma95] E. Gamma. R. Helm. R. Johnson. J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company. 1995.

[Gator02] Gator Open Source Travel System. http://animaniac.sf.net Verified 5/15/02.

[Hinckley] K. Hinckley. R. Pausch. J. Goble. N. Kassell. A survey of Design Issues in Spatial Input . Proceedings of ACM Symposium on User Interface Software and Technology. Marina del Rey, CA. 1994. 213-22

[Mario64] Nintendo Corporation, Super Mario 64 , http://www.nintendo.com/games/gamepage/gamepage_main.jsp?game_id=249

[Meinert02] K. Meinert. Travel Systems for Virtual Environments . Open Source Virtual Reality. IEEE VR 2002 Course Notes. 2002.

[Moller99] T. Moller. E. Haines. Real-Time Rendering. A K Petres. 156. 1999. A K Petres, Ltd..

[Stoakley95] R. Stoakley. M.J. Conway. R. Pausch. Virtual Reality on a WIM: Interactive Worlds in Miniature . Proceedings of ACM CHI. 1995. 265-272.

[Ware90] C. Ware. S. Osborne. Exploration and Virtual Camera Control in Virtual Three Dimensional Environments . Computer Graphics. Proceedings 1990 Symposium on Interactive 3D Graphics. 24. 2. March 1990. 175-183.