

Travel Systems for Virtual Environments

Course Notes for Open Source Virtual Reality - IEEEVR2002

Kevin Meinert

\$Date: 2002/03/04 07:04:54 \$

Table of Contents

Introduction	1
Why Travel?	1
How is Travel Used	2
Control of Travel Methods	2
The Best Travel Methods	2
Wayfinding?	3
Software Components Of A Travel System	3
Avatar	3
Virtual Camera	3
Animation	4
Collision Detection	6
Input	6
Travel System Interface	7
Integrating a Travel System into an Application	7
Conclusion	8
References	8

Introduction

This section gives an introduction to travel in virtual environments (VE) and introduces software design and implementation ideas to build a travel system. We will explain the concepts of travel in VEs at a high level so that a beginning programmer can get an idea of how to implement travel in a VE application. A travel system can be implemented with varying degrees of complexity. Affecting this complexity are factors like built-in features, reusability, and extensibility. We don't show a lot of source code. To allow these notes to stand on their own, we instead give background and descriptions of how travel methods are built and integrated into existing applications.

Navigation is composed of many elements, two of which are travel [Bowman98] and wayfinding [Darken96]. Travel methods are used to control motion of the user's viewpoint in a three dimensional environment. Wayfinding methods are used to determine a path through the environment. Some application domains that need navigation include simulation, training, and first person games. Effective navigation is critical, it can make or break the usability of a Virtual Environment (VE) application. In this paper, we are only going to talk about the *travel* part of navigation.

The concepts discussed here can also apply to 3rd person control, even though certain groups of Virtual Reality (VR) users focus mainly on 1st person travel methods.

Why Travel?

Travel allows users to gain access to areas or regions of their environment otherwise not available to them. This is, of course, also true in real life. Humans travel every day using methods such as walking, driving, flying, clicking, and typing. In VR and other 3D interactive applications 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.

How is Travel Used

Some basic examples of travel methods, this list is by no means comprehensive:

- 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 tall buildings or other multidimensional data to view it at convenient angles.
- iv. Zoom in to do fine detailed editing and then zoom out to examine the work as a whole.

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.

Control of Travel Methods

For control of motion there are some methods that have been defined, such as eyeball-in-hand and scene-in-hand [Ware90], world-in-miniature [Stoakley95], raycasting (target selection) [Hinckley], and leaning [Fairchild93]. Another common method is simple pointing, where holding a button moves the camera in the pointed direction.

- *Eyeball/Scene-in-Hand*: a way to control movement of the camera or world geometry by using a scaled down representation of the VE that fits in or near the user's hand.
- *World-in-Miniature*: a way to visualize a world view of the entire VE. Shows a scaled down version of the VE for the traveler to make decisions while in the VE.
- *Raycasting*: Used to select targets for navigation and locomotion.
- *Leaning*: A method of expressing direction of travel through body movements.

The Best Travel Methods

With regard to interaction techniques in general, [Bowman99] states: "there is no set of techniques which will maximize performance for all applications and domains". There is not an absolute best travel method or group of methods. The "best" method is application dependent, and sometimes even user dependent. For example, teleportation has the advantage of being very fast and accurate way of getting to a destination, but also can have the side effect that it can disorient the user. In some applications teleportation makes sense, in others it wouldn't. This is the same for any travel method. You need to understand your user's abilities, experiences, and expectations. Using this knowledge will help you create travel methods that are intuitive to control and unsurprising for maximum user comfort. Going back to the teleportation example, for some people teleportation could be made less disorienting when accompanied by an obvious visual queue that means "teleporter" - like a shimmering doorway or floor section. Another way to reduce disorientation is when the person knows what to expect prior to teleportation.

Wayfinding?

Wayfinding, or the process of determining a path through an environment, is a different concept from travel. It is a way of targeting a path given some set of obstacles. There are many ways to wayfind, or navigate, through a virtual environment, but we won't cover them here. [Darken96] is a starting place if you want to learn about wayfinding.

Software Components Of A Travel System

Certain software pieces need to be available to a VE application to support travel methods by the VE user. Some of the components may be available packaged together, some may be decoupled from each other, and some may have to be written, refactored, or extended to support special needs. Here we discuss a few of the necessary software components for supporting travel in an application. We wait to discuss rendering of the navigation in the section called “Integrating a Travel System into an Application”.

Avatar

Travel in VR always moves an avatar, which is at least a data representation of the person navigating, and sometimes partially or fully rendered in the VE. Whether the avatar gets rendered in the VE depends on a few things like camera placement in the travel method, and the task the user is expected to do. Avatar visibility can be set based on whether it occludes the user's view, or also whether it will be important to see it. It is important to have this avatar object even in the non-rendered case for use in keeping the avatar's state information such as position, rotation, velocity, mass, and size. The avatar state is then used for operations such as control, animation, rendering, and collision detection.

Some examples of avatar visibility determination are: when picking or placing items it can be helpful to render the avatar's arm doing the work so that the user can correlate the avatars movements to the VE; if the camera is set to be from the 1st person then some or none of the avatar would be rendered since the camera is inside of it.

This avatar data representation is also something that can be shared with other connected applications in collaborative virtual environments (CVEs).

Virtual Camera

A camera is something that manipulates a viewpoint. It 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. In some non-generalized travel systems, where the travel method has the camera attached to the avatar, the avatar and camera can be condensed into one single transform.

In an application with travel, the camera is often attached to the avatar using one or more methods described in the section called “Animation”. This attachment could be some direct offset of the avatar's attributes (useful for 1st person viewing of the avatar), or some dynamic effect to allow more advanced viewing abilities (useful in 3rd person viewing of the avatar). In short, there are many ways to attach a camera to the avatar.

Move The World, Not The Camera

This camera we've been discussing is actually a "virtual" camera, and is never actually rendered. The "real" camera is then used to affect your viewport transform, and is inversely related to this virtual camera. When the real camera is rendered, it does not use the virtual camera transform as is. It actually uses the inverse. This is very counter-intuitive for most new programmers, so we'll try to make a simple explanation.

The camera in a travel system actually has two representations, each in a different coordinate system, one that is in world space, and one in camera space. The world space camera allows intuitive animation operations in the same coordinate frame as the avatar and world geometry, while the camera space camera is simply an inverse copy of the other used for setting the viewport transform.

To completely understand, we should think of coordinate spaces [watt93], [watt92], namely the world and camera coordinate spaces. Think of a camera in each one: a world-space camera, that operates and is controlled in intuitive world coordinates; and a camera-space camera, which is always at 0,0,0 and needs things brought to it (like your world). This camera-space camera (or "moving the world") is how graphics systems actually work, and is necessary to create the effect of travel over distance (or rather, to make things move at you, since this is how it really works). Since it is unintuitive to manipulate the world to control the camera, we'd like to actually manipulate a world-space version of the camera, then when we render, convert the world-space camera to a camera-space transform of the world by simply inverting the camera's transform.

For an example, let's consider the simplest case where your avatar and virtual camera transforms are equal. The avatar is placed within the world geometry using its transform. Then, to render the camera so it appears that we are looking from the avatar, we need to bring the world from the virtual camera's location to the real camera. To bring the world to this camera, use the inverse transform of the virtual camera. This second "real" camera has this unintuitive inverse transform from the avatar because it is not really in the world space, it is the world that is in the camera's space.

Animation

To move the avatar and camera around the VE space, we need some way to update their transforms. The fact that we do viewpoint *motion* control suggests that we need some system to manage this motion. An animation system of some form is used for the viewpoint motion control. Here our definition of animation system may seem a little loose, basically we mean some controller that can influence transforms over time where the end result of these transforms may or may not be used for rendering.

The animation system can vary in complexity from fully featured supporting many reusable, extendable, and configurable animation methods to more single purpose or monolithic methods. The flexibility of this component can directly affect the flexibility of the overall travel system. An animation system also supports the concept of operators. An operator is an extractable concept that performs a specific action (two examples could be "apply force of gravity", or "set position to input device"). Animation system operators could be hand expanded and interwoven into one core or, for more generality, dynamically pluggable.

The choice for static vs dynamic operators can be made after considering performance, extension/reconfiguration, and maintainability. Extension/reconfiguration inevitably happens when it is determined that the travel system must behave in a different way. Maintainability is important to consider so that code is easy to understand for bug fixes or refactor for performance and design changes. Methods of animation are implemented with one or many of these operators modules.

Methods of Animation

We mentioned that the animation component in a travel system can be, or can be composed of, different control methods which update the avatar. Triggers, physical effects, input device, and other procedures can all play a part in the travel system making an influence on the avatar.

Direct

In direct animation the animatable values can be directly manipulated through some input device (mouse/trackball), procedure (sin wave), or by path interpolation (keyframing). Examples of these methods are:

- Direct Procedural. control of position/rotation parameters is done via self contained procedure like sine, cos, or more complex procedures like keyframing.
- Keyframing/Motion Control [watt92]. This is a procedure that iterates over some externally supplied data vector. The keyframe motion controller iterates over the elements in the vector, possibly interpolating between each element for a smooth effect over time. The interpolation can be done in many ways (i.e. linearly; on a curve fit to the data). The keyframer key type for travel is a position and rotation pair.
- Direct linkage to input device. This is useful when you want to correlate movement with some physical device. Examples are rotation of viewpoint with a knob, trackball, or mouse axis; translation of viewpoint with mouse, PDA or tablet surface.

Dynamic

In dynamic animation, the animation values are updated indirectly. For example when using Newtonian physics, animation parameters (position and rotation) can be affected indirectly through kinematics[1] or dynamics[2]. Other indirect influences can come from blasts of (virtual) air (force), stepping on a teleportation pad (position set), or getting hit by a car (collision force). Examples of dynamic control methods are:

- Indirect Procedural. Control of higher order effects like velocity, acceleration, force, are done with some procedure. For example a jump could be implemented with a sine function or some set of keyframes defined by an artist. This would influence the avatar transform additively through deltas, rather than directly.
- Indirect Linkage to Input Device. This can be useful if you want to share control of your viewpoint with other control methods. It allows input device data to affect changes (or offsets) in the avatar, rather than set specific attributes. A good example is to use keyframing to control automatic movement over terrain combined with a trackball (two axis) controller to control fine grained movements outside the preset path.
- Newtonian Physics. Particle (and rigid body) systems are a form of animation [Moller]. Modeling travel methods with Newtonian physics can give a very realistic feel. Sometimes real isn't always the easiest to control, so this should be considered.

Many things can be modeled with Newtonian physics, so this animation method can be very flexible, especially when mixed with the others. See Table 1 and Table 2 for iterative equations that implement Newtonian physics. The results calculated are changes that can be added onto current position and rotation to iteratively animate them over several frames.

Table 1. Kinematics - velocity/rotation. Compute change in position and rotation for 1 time unit. Normally these delta values are then scaled by the last frame's time measurement and then added onto the avatar.

• $dx/dt = v$	x - position (point)
• $dq/dt = 1/2 w q$	v - velocity (vector)
	q - rotation (quaternion)
	w - angular velocity (vector)
	$d?/dt$ - change in the value "?" over one time step

[1] The branch of mechanics that studies the motion of a body or a system of bodies without consideration given to its mass or the forces acting on it

[2] The branch of mechanics that is concerned with the effects of forces on the motion of a body or system of bodies, especially of forces that do not originate within the system itself.

Table 2. Dynamics - force/torque. Compute change in position and rotation for 1 time unit. Normally these delta values are then scaled by the last frame's time measurement and then added onto the avatar.

• $dx/dt = p * inv_m$	x - position (point)
• $dq/dt = 1/2 w q$	inv_m - 1.0 / mass (scalar)
• $dp/dt = F$	p - linear momentum (vector)
• $dl/dt = T$	q - rotation (quaternion)
	w - angular velocity (inv_inertia_tensor * l) (vector)
	F - force (vector)
	l - angular momentum (quaternion)
	T - torque (quaternion)
	d [?] /dt - change in the value "?" over one time step

Collision Detection

A collision detection system is something that given a set of geometry and an avatar (or some object), returns collision information. Collision detection by itself is a broad topic, ranging from navigation to computer games to robotics. You can find information about it in computational geometry, graphics, and game references. A collision detection system would be used by the collision/response operators which are plugged into the animation system. A classic case for collision detection is some set of rooms with obstacles. Using some input device to control movement, the user must not be allowed to walk through any objects or walls. To add to this, there can be iteration with the obstacles such as friction and elasticity (bounce), each of which can be adjusted from little to full effect. Another classic case is driving simulations where there is some terrain that the vehicle must not fall through.

Collision detection is a difficult issue, especially with complex scenes. Implementing a system for collision detection can be non trivial. Brute force methods of detecting collisions (those that test every object against every moving object without regard to spatial ordering) can bring a complex VE's frame rate to a crawl. Therefore it can be very important to spatially subdivide the VE into quickly accessible regions.

To integrate collision detection into the travel system we're describing, an animation operator would need to be built. The Collision/Response animation operator basically asks the collision detection system whether a collision occurred between the avatar and the scene in a given amount of time. If any collision occurred, then it determines an appropriate response to feed back into the animation system. This feed back can control another operator, but usually applies the response directly to the avatar. This response can be applied directly to position and rotation, or indirectly through the other parameters. Without impulse forces, a combination of indirect and direct update methods may need to be used [baraff98].

Input

An input system is something that gives data useful for controlling the travel system. This system is an abstraction on top of hardware, and sometimes software GUI widgets or other control signals. An input system is used by various input operators which 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. The input operators for example can apply to the

avatar forces and torques, increase velocities, or set position. Input operator design can be monolithic (all in one), or several decoupled pieces that can be mixed and matched.

There are many practical and creative ways to use the data gathered from input devices. To give a small example, consider a VR application that can run in multiple devices: desktop, head mounted display (HMD), and enclosed large screen immersive system (LSIS). For the desktop a keyboard, mouse, and possibly a head tracker may be used. For the HMD a head tracker combined with either a mouse, trackball, glove, or wand would be appropriate. For the LSIS there is a head tracker combined with wand or glove. To make it more complicated, in the desktop and LSIS cases the head tracker controls perspective while the HMD head tracker controls viewpoint rotation. In LSISs there is the added possibility the walls of the device could be only partially enclosed meaning that they might need the wand or glove to rotate the world geometry relative to the display device. All of these input and display device combinations make writing a flexible travel system interesting. It helps to have a well defined input abstraction layer defined.

Travel System Interface

To travel within an application the human user must be able to interact and control the travel method as a whole. This means the travel system should present a facade (or interface) with which the application can interact. This facade could be anything from a monolithic singleton, to several modular objects. It could even have multiple layers if for example it had several modular objects under a monolithic singleton. We've suggested a few ways, but ultimately the interface is up to the travel system software architect.

VR applications have the potential to run in several system configurations. We would like to run anywhere from a desktop to head mounted display (HMD) to large screen immersive systems (LSIS). Since systems used to display VEs have a diverse range of input devices, and because input can even come from application generated signals, a flexible travel system could provide some way to configure or route these control signals.

One way to implement this configuration flexibility of the travel system is through the animation system. If the animation system is flexible enough, it could support dynamic addition of operators. Therefore the core of the travel system would revolve around the animation system. To configure the travel system, specific operators are added to define the intended travel method.

The travel system interface can be statically or dynamically bound to the underlying animation, camera, and avatar component. If flexibility is needed, then dynamic binding can allow you to reconfigure the travel system for many application domains. This flexibility would allow the travel system to be placed in a reusable library for use by many types of VEs.

Integrating a Travel System into an Application

At first it may seem natural to build the travel system directly into the application main computation loop. As the application grows the need to refactor and decouple components becomes more important. For the simplest of applications and travel methods this ad-hoc approach will probably be quite effective. For more complex applications it could be wise to use a travel system that is external or at least decoupled from the main application code. In other words, it would be nice if the travel system were a reusable component able to be dropped into any library.

The application will usually need to update the travel system through some `update()` or similar call. The application may need to render the appearance of the avatar, and will need to set the viewport transform to the travel system's camera. See the section called "Move The World, Not The Camera" for details on getting the correct camera transform to pass to the rendering system.

Conclusion

This paper has outlined what travel is, some travel methods, and what it takes to create software to implement those methods. 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 a travel method for their VE, they should be able to focus wholly on the biology problem at hand. This paper has hopefully given you a high level description of travel system software, and ideas for implementation.

References

- [baraff98] David Baraff. Andrew Witkin. "Physically Based Modeling". *Siggraph 98 Course Notes*. 1998. this paper is available on-line [<http://www.cs.cmu.edu/~baraff/sigcourse98>].
- [eberly2001] David Eberly. "Collision Detection ch.6". *3D Game Engine Design*. Morgan Kaufmann Publishers. 2001. Copyright © 2001 Academic Press.
- [Moller] Tomas Moller. Eric Haines. *Real-Time Rendering*. A K Petres. 156. 1999. Copyright © 1999 A K Petres, Ltd..
- [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.
- [Darken96] Rudolph Darken96. John Sibert. "Wayfinding Strategies and Behaviors in Large Virtual Worlds". *ACM CHI*. 1996. 142-149.
- [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.
- [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.
- [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 ReyCA. 1994. 213-222.
- [watt93] Alan Watt. *3D Computer Graphics (second ed.)*. Addison Wesley Publishing Company Inc.. 1996. 10. Copyright © 1993 Addison Wesley Publishing Company Inc..
- [watt92] Alan Watt. Mark Watt. *Advanced Animation and Rendering Techniques*. Theory and Practice. Addison Wesley Publishing Company. 1995. 6, 337, 345-368, 414. Copyright © 1992 ACM Press.