jReality — Interactive Audiovisual Applications Across Virtual Environments

Peter Brinkmann∗
The City College of New York
Charles Gunn†
Technische Universität Berlin
Steffen Weißmann‡
Technische Universität Berlin

Abstract

jReality is a Java scene graph library for creating real-time interactive applications with 3D computer graphics and spatialized audio. Applications written for jReality will run unchanged on software and hardware platforms ranging from desktop machines with a single screen and stereo speakers to immersive virtual environments with motion tracking, multiple screens with 3D stereo projection, and multi-channel audio setups. In addition to euclidean geometry, jReality supports hyperbolic and elliptic geometry.

jReality comes with a number of graphics rendering backends, ranging from pure software to hardware-accelerated to photorealistic. A distributed backend is available for cluster-based virtual environments. Audio backends range from a basic stereo renderer to a high-performance Ambisonics renderer for arbitrary 3D speaker configurations.

jReality achieves device-independent user interaction through a layer of abstract input devices that are matched at runtime with available physical devices, so that a jReality application will work with keyboard and mouse in a desktop environment as well as with motion tracking in a virtual environment.

CR Categories:

Keywords: Virtual Reality, Desktop Virtual Reality, Immersive Virtual Environments, Java

1 Introduction

jReality is a visualization and sonification library for virtual reality applications. It facilitates the creation of interactive three-dimensional audiovisual scenes that are portable across a wide range of hardware and software platforms. jReality is open source software, covered by a BSD license. The code and extensive documentation are available at http://www.jreality.de. jReality is the primary software platform of the PORTAL at the Technical University of Berlin and the VisorLab at the City College of New York. It was featured in Imaginary 2008 http://www.jreality.de an interactive exhibition of mathematical visualization.

jReality differs from other scene graphs in terms of scope and flexibility. Most importantly, jReality supports a wide range of hardware configurations; a jReality application will run unchanged on platforms ranging from desktop machines with a single screen and stereo speakers to virtual reality environments with motion tracking, multiple screens with 3D stereo projection, and multi-channel audio.

jReality graphics backends include a software viewer that can be deployed as an applet or WebStart application. An OpenGL backend will take advantage of hardware accelerated rendering, and a distributed backend will display jReality scenes on virtual environments with multiple screens. Similarly, audio backends will render a stereo signal through JavaSound when running on a desktop system but may also render Ambisonics[Gerzon 1974] through the JACK Audio Connection Kit http://jackaudio.org when running on a multi-channel system.

User interaction in jReality separates the meaning of the interaction from the hardware of the interaction. For instance, a scene may allow the user to rotate an object in the scene. The designer of the scene attaches a rotation tool to the object without considering how the user will effect a rotation. At runtime, the tool system determines what input devices are available and matches them with the tools in the scene. On a desktop computer, a rotation will typically be caused by a mouse dragging event. In a virtual environment, motion tracking or wand gestures may trigger a rotation. The same application will run in a wide range of different environments, without requiring any changes.

Finally, jReality is metric-neutral, providing seamless support for hyperbolic and elliptic space through its reliance on the projective models of these spaces, making these important mathematical concepts available to a wide audience.

2 Related work

We review related work in the context of portability, metric-neutrality and audio functionality.
VTK, OpenSceneGraph\footnote{http://www.openscenegraph.org} and OpenSG\footnote{http://sourceforge.net/projects/osgal/} are flexible, OpenGL-based software packages for 3D computer graphics. Each of them consists of a graphics library and bindings for a number of languages, including Python and Java. Their size and use of a C++ library, however, complicate installation and preclude the deployment of applications through the web. These libraries have no built-in support for distributed immersive environments and 3D input devices, but there are third-party libraries providing this functionality, for instance via VRJuggler. Non-euclidean geometry and spatial audio are not supported, but there exist libraries providing such functionality\footnote{http://vrjuggler.org/sonix/} [Neumann et al. 2003].

VRJuggler [VRJ] is a development framework for portable VR applications. It does not provide a scene graph or any graphics API. Instead, VRJuggler applications can choose from various graphics APIs (OpenGL, Open Scene Graph, VTK, etc.), VRJuggler will set up projection matrices and perform rendering. Basic support for spatial audio is available via the Sonix\footnote{http://connect.creativelabs.com/openal/} module, which provides an interface to standard libraries such as OpenAL, which are limited to stereo and surround sound. Like Syzygy, VRJuggler expects input from tracking devices and provides a simulator for desktop usage.

Syzygy [Schaeffer and Goudeseune 2003; Schaeffer et al. 2005] offers a scene graph framework that is a thin overlay of OpenGL/GLUT functions. Written in C++, with Python bindings, it is highly optimized for use in distributed, CAVE-like installations. Syzygy expects input from tracking devices; if no tracking devices are available, users have to communicate with Syzygy through a tool that simulates trackers. Spatial audio support is provided using libraries such as FMOD\footnote{http://openal.org} or OpenAL\footnote{http://www.openal.org}\footnote{http://connect.creativelabs.com/openal/}. These libraries support only stereo and surround output.

Geomview is an interactive 3D viewer for Unix\footnote{http://www.geomview.org}. Like jReality, Geomview is rooted in the mathematical visualization community. It was a pioneer in the implementation of metric-neutrality. It is written in C and, like the previous entries, is not easy to deploy on the web. Geomview has no audio support and is not intended for use in immersive environments.

Java 3D\footnote{http://java.sun.com/products/java-media/3d/index.html} [Selman 2002] allows development of interactive 3D applications for different environments which can be deployed via the standard Java mechanisms (Applets, Java Web Start). In Java3D, interaction (which is modelled by Java3D Behaviors) is directly wired to the user input devices on a certain environment. So porting applications to different environments requires changes in the applications source code. Since the establishment of the first Java3D CAVE\footnote{http://www.jabiru.com/} [Sensen et al. 2002], together with the development of the Java3D library Jabiru [Stromer et al. 2005], Java3D behaviors can be abstracted (using a ConfiguredUniverse) such that VE applications can be tested and run on a desktop without code modifications. Jabiru is also focused on using desktop computers for developing and testing VE applications, rather than using a desktop environment as a fully fledged virtual reality environment. With Jabiru, it is also possible to publish VE applications via Java Web Start for desktop users. Java 3D supports spatial audio with stereo rendering. An extension for true spatial audio output on speaker arrays using vector based amplitude panning (VBAP) appeared in [Sheridan et al. 2004], and Java3D was used with Ambisonics in a virtual environment [Schiemer et al. 2004].

3 Design of jReality

jReality provides a clear separation between functionality:

The scene graph describes the 3D scene including geometry, appearance attributes, audio sources as well as tools, which define interaction with the 3D scene.

Rendering backends convert a reality scene graph into graphics. There are various backend for interactive and non-interactive rendering.

The tool system triggers the tools attached to the scene graph based user interaction via input devices.

Audio backends render the spatial audio of the scene. There are different backends for different speaker setups, for instance stereo, 5.1 surround or ambisonics for arbitrary speaker setups.

3.1 The jReality scene graph

A jReality scene graph is a directed graph without cycles. A scene graph is not necessarily a tree because an instance of a scene graph node may appear in different locations of the graph at the same time. The location of a node in the scene graph is specified by a path to the node.

Rendering backends are derived from the abstract class Viewer. Viewers take two parameters: a scene graph root node and a path that specifies the currently active camera. Note that the root node is not an intrinsic property of the scene graph but a choice for rendering purposes.

In jReality, all nodes of the scene graph are derived from a common abstract superclass: SceneGraphNode. SceneGraphComponent is the only one subclass of SceneGraphNode whose instance can have descendants. All other nodes are leaves. The leaves in detail:

Transformation: a leaf node of the scene graph that specifies a local coordinate system via a 4x4 matrix. This class supports transformations in euclidean, spherical and hyperbolic geometries. The transformation from object coordinates to world coordinates is the composition of transformations along a path from an object node up to the root node.

Appearance: a leaf node of the scene graph that represents shading attributes such as colors and textures as a key-value dictionary. Keys must be strings; values can be one of several standard types or an arbitrary Java class instance. There is no fixed list of attributes; different viewers may honor different sets of attributes. Shader names themselves are part of this dictionary, so that different shaders can be specified.

While each rendering backend is responsible for implementing a standard set of shaders not all backends are expected to implement all shaders. For example, the OpenGL backend supports shaders written in the OpenGL Shading Language\footnote{www.opengl.org}, and shader parameters for RenderMan, will only make sense for certain viewers. A backend that encounters an unknown shader falls back on one of the standard shaders.

When traversing a scene graph, viewers are expected to maintain a stack of appearances encountered on the way from the
root node. The value of the dictionary at a specific node in the scene graph is then controlled by an inheritance mechanism which depends on the collected appearances along the path from the root through the scene graph to this node.

**Geometry:** a leaf of the scene graph that holds geometry information, such as face sets, Bezier patches, etc. Instances of this class can store attributes beyond the usual coordinates and connectivity information.

As in the case of appearances, the set of possible attributes is not fixed, and again stored in dictionaries (separate ones for vertex, edge, and face attributes). This way data for non-standard shaders (like a per-vertex temperature information) can be stored directly on the geometry using the appropriate key. There is no inheritance for geometry attributes.

**Camera:** a leaf that holds a camera. Every jReality viewer contains a scene graph root node, and a camera path. The latter is a scene graph path that begins with the root and ends at a camera node. Rendering for this viewer is always with respect to this camera. Cameras are general enough to support rendering in a variety of environments, including stereo and off-axis alignment (such as CAVEs).

Attributes of an instance of `SceneGraphComponent` include a geometry, a transformation, an appearance, a camera, and a light, each of which may be null, as well as a list of descendants of type `SceneGraphComponent`. For every leaf in a connected scene graph there may be more than one path from the selected root node. Each such path defines one copy of the leaf when the graph is traversed.

Further attributes of an instance of `SceneGraphComponent` are flags indicating whether geometries attached to this instance and its descendants are visible, (resp. pickable), and a list of tools that provide interactive features associated with this instance.

### 3.1.1 Internals

The clean separation of frontend and backend is achieved by means of the visitor pattern [Gamma et al. 1995]. The visitor pattern allows for the addition of different backends without the need to modify or extend the classes that are of concern to application programmers.

Thread-safety is achieved by means of a read/write lock mechanism and that allows users to wrap several sequential changes into a Runnable and execute them as if they were atomic.

Notification of changes to the scene graph are communicated by means of a system of events (AppearanceEvent, CameraEvent, GeometryEvent, etc.). Any change of the scene graph is reported by firing the appropriate events. Some backends use these events to maintain copies of scene graphs. The OpenGL backend does so for rendering reasons, while the CAVE viewer uses this mechanism to keep several copies of a scene graph synchronized across a network.

### 3.2 Rendering backends

jReality graphics backends include a pure Java software renderer and a hardware accelerated OpenGL renderer. The former offers superior quality particularly for transparent scenes and can also be deployed remotely where native extensions (required by the OpenGL backend) are not allowed. In cluster-based virtual environments, a distributed backend will display scenes on multiple screens.

Moreover, jReality comes with a number of noninteractive backends, including a backend that exports scenes to PDF for inclusion in publications as well as a backend that exports scenes to Pixar’s RenderMan for high-quality batch processing. It includes also a U3D backend, which allows one to embed interactive 3D content in PDF documents.

jReality reads many popular 3D file formats (OBJ, 3DS, STL, VRML 1.0, Mathematica Graphics 3D, JVX), and writes many of the same, enabling easy exchange of data or output to 3D printers.

Graphics rendering is controlled by appearances, a mechanism for defining and inheriting properties such as parameters for the point, line, and polygon shaders. jReality shaders can also be customized for individual backends, for instance GLSL shaders for the OpenGL backend.

### 3.3 The jReality Tool system

The tool system of jReality separates the meaning of user interaction from the hardware of the interaction. For instance, a scene may allow the user to rotate an object. The designer of the scene attaches a rotation tool to the object without considering how the user will effect a rotation. At runtime, the tool system determines what input devices are available and matches them with the tools in the scene. On a desktop computer, a rotation will typically be caused by a mouse dragging event. In a virtual environment, motion tracking or wand gestures may trigger a rotation. The application remains the same in both cases.

Currently drivers are available for keyboard, mouse, joystick, Trackd (supporting all popular motion tracking and immersive input devices such as Ascension MotionStar or A.R.T. DTrack systems), Wii Remote, Space Navigator, and GameTrak.

### 3.4 Audio

jReality supports spatialized audio. Various audio sources (e.g., media players, hardware input, software synthesizers) can be placed in a scene. The audio rendering pipeline of jReality includes auxiliary sends and returns for inserting effects and distance cues like reverberation and distance-dependent attenuation, following [Furse 2000]. It also implicitly models sound propagation, yielding physically accurate Doppler shifts.

Audio backends will render a stereo signal when running on a desktop system, or 5.1 surround in home theater setups. On a 3D multi-channel speaker rig, jReality will render spatialized audio using Ambisonics [Gerzon 1974].

### 4 Portability across virtual environments

jReality applications are portable across desktop and immersive VE. This is possible because of the clear separation of the jReality scene graph and the rendering backends. On a desktop one would usually use a jogl-Viewer, which displays the jReality scene in a desktop window, using hardware accelerated OpenGL rendering. In an immersive multi-wall VE, one uses the distributed CAVE backend, which will render the same jReality scene graph in the distributed immersive VE.

Portable interaction is achieved by the jReality tool system, which is based on the idea of virtual devices, a set of devices that is independent of the current environment and its hardware. The available hardware in a specific environment is mapped to a common
set of virtual devices, which is done in an environment-dependent configuration file. The jReality tools, which are responsible for modifying the jReality scene graph depending on user input, are triggered via these common virtual devices. While this approach is obvious for devices like buttons or joystick axes, the definition of common virtual devices for the interaction with the 3D scene (e.g. by the mouse pointer, or a wand in an immersive VE) requires further explanation and a precise definition of the environments we want to address. The original intention was to develop software that runs unchanged in a CAVE-like environment (with head-tracking and a tracked wand) and in desktop setups with the following input devices:

- Mouse and Keyboard
- Joystick
- Wii Remote
- Space Navigator
- SensAble PHANTOM Omni Haptic Device

We achieve the same degree of immersion on desktop VEs by using first-person gaming techniques. While the user in an immersive VE may walk/turn and look around freely within the physical bounds of the environment, a user running the same application on a desktop can move and look around as in a first-person 3D computer game. In this way, no simulator of tracking devices is required. Therfore a transparent navigation model is required, which works for both immersive and desktop VEs.

On a desktop, navigation can be understood as transforming the transformation of the camera node due to user input. In an immersive environment, this is not possible since the camera node is transformed due to head-tracking. To obtain a navigation model that works in both cases, we introduce an extra parent node of the transformed due to head-tracking. To obtain a navigation model rungs unchanged in a CAVE-like environment (with head-tracking and a tracked wand) and in desktop setups with the following input devices:

- Mouse and Keyboard
- Joystick
- Wii Remote
- Space Navigator
- SensAble PHANTOM Omni Haptic Device

We achieve the same degree of immersion on desktop VEs by using first-person gaming techniques. While the user in an immersive VE may walk/turn and look around freely within the physical bounds of the environment, a user running the same application on a desktop can move and look around as in a first-person 3D computer game. In this way, no simulator of tracking devices is required. Therfore a transparent navigation model is required, which works for both immersive and desktop VEs.

In an immersive VE we navigate the physical space inside the VE through the 3D scene, like a space ship. The head-tracked user can freely move and look around inside of the space ship. The head-tracking transformation is directly applied to the head component. On a desktop, we imagine an avatar that is walking, jumping or flying around. The avatar component corresponds to the avatar’s feet. The head component may move relatively to the avatar’s feet, for instance as a default it is 1.7 meters above. But the avatar might crouch and look up and down, which is the desktop equivalent of head-tracking. This artificial tracking transformation is for instance changed due to mouse look. To summarize: Navigation is applied to the avatar node. The head is transformed relatively to the avatar node, for instance by head tracking or by mouse look.

For interaction with objects in the 3D scene, each environment is assumed to have a designated pointer device. It is used to select an object for interaction, and its motion will usually determine how the selected object will be modified. The corresponding virtual device is a transformation matrix (the PointerTransformation), whose translation determines the location of the pointer device, the x-axis determines the pointer direction, and the x- and y-axes determine the orientation. In an immersive environment, this device is the tracking transformation of the wand. In a desktop environment (without a 6-DOF input device), the pointer device is obtained from a point on the screen, which may be the mouse pointer or a crosshair at the screen center. Such a point defines a ray in the 3D scene, starting at the near-clipping plane and points towards the view direction. This defines the translation part of the PointerTransformation as well as its z-axis. The x- and y-axes are computed in such a way that they are as close as possible to the screen coordinate axes while forming a ONB together with th z-axis.

In this way we have defined a natural replacement for the 6-DOF wand input device which is available in an immersive environment. It is obtained by a suitable lift of the 2D mouse position to a 3D transformation. Of course it is not possible to achieve the six degrees of freedom of a wand with the 2D mouse input, but in combination with the first-person navigation, we obtain a 5-DOF pointer device. The only missing degree of freedom is the rotation about the pointer axis. It is possible to implement control of this last DOF using a modifier key or similar, but it turned out that the 5-DOF pointer feels very natural and does not cause serious restrictions in our applications.

jReality allows to immerse 2D Java Swing GUI into the 3D scene. In this way no additional effort is required to provide access from within the immersive environment to functionality that is controlled via the applications GUI.

4.1 The jReality Tool system

The tool system of jReality creates device drivers and virtual devices based on a environment dependent configuration file. Device drivers are called raw devices, while all other devices are virtual devices. Each axis or transformation of a device is mapped to an input slot, which makes it available to virtual devices and tools.

4.1.1 Virtual Devices

A virtual device provides one single axis state or transformation matrix as output. As input, it takes one or more axes and transformations from other virtual or raw devices. The input and output slots of a virtual device are specified in the tool system configuration file. The devices report state changes via an event mechanism. One special class of virtual devices come from the 3D scene itself.

4.1.2 Tools

A tool is an object attached to the scene, which is registered to react on certain input slots. We distinguish tools depending on whether they require a geometry being pointed at to become active (pointer tools) or not (always-active tools). Examples for pointer tools are dragging and rotate tools. Typical always-active tools are navigation tools and head-tracking tools. A tool carries a fixed list of activation slots, which is empty for always-active tools. A tool also
maintains a (varying) list of current slots, indicating to which slots it reacts on while it is active. The basic interface of a tool consists of the three methods `activate`, `perform` and `deactivate`. A tool is active while at least one axis state of the activation slots corresponds to the state of a pressed button. This state is signaled to the tool via the method calls `activate/deactivate`. While the tool is active, the `perform` method is called whenever the value corresponding to a current slot changes. To access relevant data from inside a tool, a `tool context` is passed to the tool methods, providing access to the states of virtual devices and to the 3D scene.

jReality includes tools to modify a scene, like the Rotate-, Scale- and DraggingTools, which can be plugged into any scene for user modification. The DragEventTool allows registering listeners that will be notified about drags (in object coordinates) with the pointer device, on the level of points, edges, faces or geometries. The listeners can then modify the geometry of other parts of the scene depending on the drag events. Besides these pointer-based tools, jReality also includes navigation tools, like the ShipNavigationTool which allows terrain following as in a first-person computer game, or a FlyTool. Tools coming with jReality are implemented in a metric-neutral way. They will work in spherical and elliptical space just as in euclidean space.

5 Audio

6 Metric neutrality

jReality is metric-neutral, a possibility discussed in [Phillips and Gunn 1992] and implemented in Geomview[Muñzner et al.]. In jReality, the metric can take one of the three values euclidean, hyperbolic, or elliptic. The metric is defined in the scene graph via an `Appearance` attribute; by default it is euclidean. Tools and other elements of the scene graph can adjust their behavior to the ambient metric by querying the appearance inheritance mechanism for the current metric. By the mechanism of inheritance it is therefore possible for the same scene to contain sub-graphs of different metric; for example, a virtual museum (modeled in euclidean space) might contain an exhibit featuring motion in hyperbolic space next to one showing elliptic space.

6.1 Infrastructure

jReality offers a full set of mathematical classes to support metric-neutral explorations. These classes typically provide static methods for doing geometric processing and kinematics in these spaces. The core space for jReality is \( \mathbb{P}^3(\mathbb{R}) \), real projective space, not \( \mathbb{R}^3 \). There are a core set of purely projective methods involving incidence properties, for example, for finding the common (possibly infinite) point of three projective planes.

Examples of metric-dependent geometric processing include distance and angle calculations involving points, lines, and planes; orthogonal projections; and curve interpolation. The standard tubing option for the default jReality line shader uses such features to create non-euclidean tubes around line sets located in non-euclidean space. See Figure 3.

Additionally there are methods for calculating projectivities and isometries as 4x4 matrices including: central projections, affine transformations, and metric isometries such as translations, rotations, reflections and glide-reflections, and screw motions (in all three metrics).

Building on this infrastructure, jReality offers some new features related to non-euclidean spaces, which are described below.

Figure 2: The jReality audio rendering pipeline.
6.2 Real-time non-euclidean shading

[Gunn 1993] reported on the use of a RenderMan shader to correctly render shaded images of elliptic and hyperbolic space. In jReality this has been extended with an OpenGL Shading Language shader which does the same in real-time. See the jReality developer tutorial class NoneuclideanExample for an example of how to use this shader.

6.3 Spherical space

As described in [Weeks], due to a fortunate design accident in the standard 3D rendering libraries, it is possible to also render spherical space, which is a double covering of elliptic space. This feature has been implemented also in jReality. It can be activated by setting the Attribute \( \text{"renderS3"} \) to true in the root appearance.

6.4 True non-euclidean tracking

By combining this non-euclidean infrastructure described above with the virtual reality features of jReality, we have been able to achieve non-euclidean tracked motion in a VR environment (VRE) for the first time. Previous work involving display of non-euclidean geometry in VRE’s (for example, [Francis et al. 2003]) had featured the ability to fly non-euclideanly through the space (typically populated with a regular tesselation), but tracking software remained euclidean. That is, when the tracked observer moved within his virtual world, the motion was expressed by applying euclidean isometries to move the virtual cameras representing the observer’s eyes. This is understandable since most VR systems provide a closed-box tracking solution that is implicitly euclidean. But the result was an unsatisfactory mixture of euclidean and non-euclidean motion.

To obtain true non-euclidean tracking, jReality models the VRE as a subset of projective space by assigning homogeneous coordinates to the geometry of the room itself and then integrating this geometry into the metric-neutral infrastructure of jReality.

There were some challenges in the process of doing this. Typically a VRE comes with a built-in tracking system based on a physical coordinate system. For example, suppose the VRE is a cube whose tracking information is in meters. Suppose a point at the center of the is the origin of this coordinate system. Then a typical corner has coordinates \((\pm 1, \pm 1, \pm 1, 1)\). If I have a data set that is much smaller or larger than this cube, in the euclidean case I can simply apply an isotropic scale to the data in order to obtain the “right size”, leaving the VRE coordinate system unchanged.

6.4.1 Non-euclidean space is not scaleable

This fixed coordinate system however is not compatible with non-euclidean tracking. Consider for example the projective model of hyperbolic space used in jReality: it consists of the interior of the unit ball. If one naively uses the coordinates provided with the tracking software, as above, then all the vertices of the virtual room lie completely outside of hyperbolic space. On the other hand, one cannot simply scale the world to compensate for this. On the simplest level, scaling does not preserve the unit ball model of hyperbolic space. There are ways to model hyperbolic space on a ball of variable radius \( r \) but these models prove difficult to coordinate with the standard model implicit in most of the infrastructure described above. More generally, isotropic scaling has much different behavior in non-euclidean space compared to euclidean space. In euclidean space, as long as everything in the world is scaled by the same factor, there is no practical way to detect the scaling: it’s conformal, so shape is preserved. Scaling commutes with isometries. In non-euclidean space, however, isotropic scaling applied to geometry, changes the shape of the geometry.

6.4.2 Changing the tracking units

Rather than scaling the world to fit to the fixed dimensions of the VRE, one can make the units returned from the tracking device more flexible. We introduced a real parameter corresponding to this change of unit length parameter and interposed it into the tracking system as a virtual device (see discussion of tool system 3.3), so that the translation part of the tracking transformation is scaled by this value. Setting it to a value of \( k \), for example, means that a typical corner of the VRE space receives coordinates of \((\pm k, \pm k; \pm k, 1)\), which is a point safely within hyperbolic space for small \( k \). In effect this scaling parameter shrinks (or expands) the virtual room. If we think of this as a “space ship” then we are in effect making the space ship smaller or larger when we change this parameter. And, according to the discussion above, as the room shrinks or expands, it changes shape! For example, in hyperbolic space with homogeneous coordinates for \( k = .2 \), the walls meet at hyperbolic angles of 87.6\(^{\circ}\); for \( k = .5 \), they meet at 70.5\(^{\circ}\). The same cases in the elliptic space yield angles of 92\(^{\circ}\) and 101.5\(^{\circ}\). For use in non-euclidean spaces, one will typically need to shrink VRE coordinates considerably in order to feel comfortable investigating the spaces. Of course the same method also works in euclidean space.

6.4.3 Implementation

Now, once this flexible coordinate system for the VRE is established, one can proceed to implement non-euclidean tracking. The euclidean transformation provided by the tracking device is factored into translation plus rotation. Let \( P \) be the image of the origin \( O = (0, 0, 0, 1) \) under the translation. One then constructs a non-euclidean translation \( T_e \) carrying \( O \) to \( P \). This translation is then recombined with the rotation to produce a non-euclidean isometry, which is then applied to the head node. The head then occupies the same projective point as it would under a euclidean translation, and has the same orientation in space. The important difference is that the scene undergoes thereby a non-euclidean isometry. For example, in hyperbolic space, as the observer backs up away from the

Figure 3: Hyperbolic, euclidean, and elliptic tubes around a horizontal line segment.
front wall, the objects on the front wall will tend to reduce their apparent size exponentially quickly. Experience has shown that such phenomena provide the observer important insights into the nature of non-euclidean geometries not easily obtainable through other means.

6.4.4 Conclusion

The non-euclidean tracking in jReality is made possible by pre-scaling the virtual environment coordinate system so that the virtual environment occupies an appropriate subset of the projective model for the non-euclidean geometry. In this way it avoids the problems inherit in combining scaling transformations with non-euclidean transformations in the scene graph, and provides for the first time the virtual experience of moving within the non-euclidean world.

7 Applications

8 Conclusion

We totally rule — once again.

Acknowledgements

Thanks to all jreality contributors etc.

References


Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.


