

REAL-TIME INTERACTIVE 3D AUDIO AND VIDEO WITH JREALITY

Peter Brinkmann

The City College of New York
Department of Mathematics
New York, U.S.A.

Steffen Weißmann*

Technical University of Berlin
Department of Mathematics
Berlin, Germany

ABSTRACT

We introduce jReality, a Java library for creating real-time interactive audiovisual applications with three-dimensional 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. We present an overview of the capabilities of jReality as well as a discussion of its design, with an emphasis on audio.

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 written in Java and will run on any common operating system.

While jReality was originally conceived as a tool for mathematical visualization [3], it has since grown to become a general platform for real-time interactive audio and video. The purpose of this paper is to advertise jReality as a tool for the creation of audiovisual works, and to give potential developers an overview of the design of jReality.

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 2008,¹ an interactive exhibition of mathematical visualization.

2. OVERVIEW OF JREALITY

A *scene graph* is a hierarchical representation of geometric objects and transformations that make up a scene. For instance, when describing a human arm, one can think of each joint as a transformation matrix, with the anatomy of

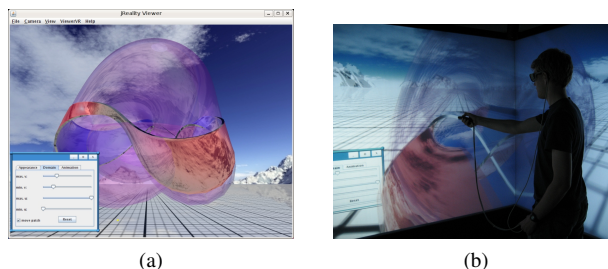


Figure 1. The same jReality application running on a desktop machine (a) and in the PORTAL, a CAVE-like virtual environment (b).

the arm giving a hierarchy of transformations. The position of the hand depends on the state of the wrist, which depends on the state of the elbow, etc. Hierarchies of this kind frequently arise in computer graphics.

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 (Figure 1).

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[2] through the JACK Audio Connection Kit² 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

* Supported by the DFG Research Center MATHEON

¹<http://www.imaginary2008.de/>

²<http://jackaudio.org/>

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.

Unlike most computer graphics libraries, jReality is *metric neutral*, i.e., it supports curved geometries (specifically, hyperbolic and elliptic) as well as the familiar flat euclidean geometry, which opens up creative possibilities that remain largely untapped. Finally, mechanical simulations are possible with the recent integration of jBullet³, the Java port of the Bullet physics engine.

3. RELATED WORK

OpenSceneGraph⁴ and OpenSG⁵ are widely used scene graph libraries written in C++. They do not have built-in support for audio, but there exist libraries providing such functionality⁶ [4]. Java3D⁷ and Xith3D⁸ are both java scene graph APIs with support for spatial audio, supporting stereo speakers and headphones for output. An extension for true spatial audio output on speaker arrays using vector based amplitude panning (VBAP) appeared in [8], and Java3D was used with Ambisonics in a virtual environment [7].

Several visualization libraries such as Amira⁹ and CaveLib¹⁰ do not provide audio support, while others (VR Juggler¹¹, Syzygy¹², and jMonkeyEngine¹³) provide audio support using libraries such as FMOD¹⁴ or OpenAL¹⁵. These libraries support only stereo and surround output. None of them match jReality in scope and flexibility.

4. PHILOSOPHY AND DESIGN OF JREALITY

For the purposes of jReality, a scene graph is a directed graph without cycles¹⁶ where each node may have a number of properties, including a geometry, a sound, a transformation, a set of appearance attributes, and a list of child nodes. Transformations describe the location and orientation of a node relative to its parent node.

³<http://jbullet.advel.cz/>

⁴<http://www.openscenegraph.org/>

⁵<http://www.opensg.org/>

⁶<http://sourceforge.net/projects/osgal/>

⁷<http://java3d.dev.java.net/>

⁸<http://xith.org/>

⁹<http://www.amiravis.com/>

¹⁰<http://www.mechdyne.com/>

¹¹<http://www.vrjuggler.org/>

¹²<http://www.isl.uiuc.edu/syzygy.htm>

¹³<http://www.jmonkeyengine.com>

¹⁴<http://www.fmod.org/>

¹⁵<http://connect.creativelabs.com/openal/>

¹⁶It is tempting to think of the scene graph as a tree, but it differs from a tree in one crucial respect: Nodes without descendants, or even entire subtrees, may appear in several places in the scene graph, avoiding needless duplication of information.

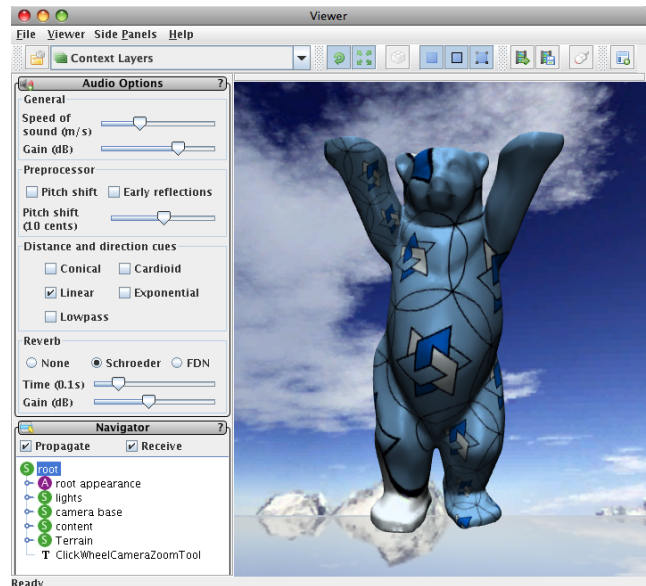


Figure 2. A screen shot of the program in Section 6, showing a jReality viewer window with a typical user interface consisting of controls for appearances and audio properties.

One core design feature of jReality is a clear separation between the scene graph (*frontend*), the rendering components (*backends*) that translate the scene graph into graphics and sound, and the tool system. jReality has been designed for *thread-safety*, so that several backends and tools can operate on one scene graph at the same time.

Real-time rendering backends for jReality include a software renderer that requires nothing but standard Java libraries, an OpenGL renderer that takes advantages of hardware acceleration on graphics cards, and a distributed renderer for virtual reality environments with multiple screens.

jReality also comes with a number of backends that are intended for batch processing, such as a backend that writes files in Pixar’s RenderMan format, suitable for high-quality rendering with ray-tracing, and a PDF backend.

5. AUDIO IN JREALITY

Designing an audio architecture that agrees with the general philosophy of jReality posed a number of challenges. The audio components had to admit a variety of inputs, provide a clear separation between audio nodes in the frontend and the rendering backend, support a range of output formats and devices, and be thread-safe so that several audio backends can operate in parallel. A usage scenario of the latter would be an application that lets several users with headphones and VR goggles act independently within the same scene.

The core element of the audio architecture is a new scene graph node called *audio source* that can be attached to a scene graph component just like a geometry or a light.

The responsibilities of an audio source are limited to maintaining a circular sample buffer, writing samples upon request, and handing out readers for its circular buffer to consumers of samples. A reader first fulfills sample requests with samples that are already in the circular buffer; when the reader reaches the current end of the buffer, it requests more samples from the audio source.

When asked to render n samples, an audio source may choose to render more than n samples. For instance, if an audio source wraps a software synthesizer, it may be convenient to render a multiple of the buffer size of the synthesizer. Our approach implicitly reconciles buffer sizes across audio sources and output devices.

The audio source keeps track of time in terms of the number of samples requested so far. An audio source generates a mono signal at the sample rate of its choosing; sample rate conversion, spatialization, and multi-channel processing are the responsibility of the backend.

The deliberately simple nature of audio sources is intended to encourage developers to implement audio sources that draw their samples from a wide variety of inputs. Our current list of audio sources includes a node that reads sound files and audio streams, a rudimentary interface for implementing software synthesizers in Java, an adapter for JASS audio sources[9], and a node that internally uses Csound¹⁷ as its synthesis engine.

Finally, we have a node that reads from ports of the JACK Audio Connection Kit. The output of most audio applications under Linux and Mac OS X can be routed through JACK.¹⁸ Many applications can be controlled by tools that generate MIDI or OSC events, so that jReality acts as a spatialization tool for a large collection of audio applications.

An audio backend traverses the scene graph, picking up a sample reader and transformation matrices for each occurrence of an audio source in the graph. For each occurrence of an audio source, it creates a *sound path* that is responsible for modeling the propagation of sound emitted by the source. It also keeps track of a microphone in the scene.

Having one reader for each backend and each occurrence of an audio source solves two of our central design problems at once: We can attach the same audio node to several different scene graph components, and we can run several backends in parallel. Only the first reader to exhaust the currently available samples will trigger the generation of new samples; all the other readers will merely reuse samples that were computed before. Backends can drift relative to each other, up to a point; if one backend falls too far behind, it will be lapped, lose about one buffer's worth of samples, and continue rendering.

Audio rendering is driven by a callback method that renders one audio frame at a time. This callback-driven design

is similar to and inspired by the JACK API, but it works equally well with other architectures such as JavaSound. In the callback method, the backend updates the current microphone position as well as the position of each sound source and then updates the sound paths with the new location.

Sound paths are responsible for most of the audio processing. A sound path first computes the current position of the sound source relative to the microphone. This is a subtle problem because transformation matrices in the scene graph will be updated a rate that is roughly the video frame rate. The video frame rate is typically much lower than the audio frame rate, so that the sound path must smooth out the location information to prevent discontinuities.

With the interpolated relative source position in place, the sound path applies suitable transformations to the audio samples, including distance-dependent delays (yielding physically accurate Doppler effects), conversion from the sample rate of the source to the sample rate of the audio hardware, plus possibly *distance cues* such as attenuation, low-pass filtering, and reverberation [1].

The *sound encoder* is the final element of the audio rendering pipeline. Given a sample and the coordinates of origin of the sample, it computes a spatialized audio signal and sends it to the audio hardware or a sound file.

The current list of encoders includes a simple stereo encoder for JavaSound, a 5.1 surround encoder for JavaSound using vector-based amplitude panning [6], a second-order planar Ambisonics encoder for JACK, and a first-order three-dimensional Ambisonics encoder for JACK. So, in line with the general philosophy of jReality, we target audio hardware ranging from typical desktop setups to home theater systems to multi-channel 3D installations like the VisorLab.

6. EXAMPLE

The code in Appendix A is a sample program that illustrates some of the major points outlined in this article. It creates a main viewer panel surrounded by an implicitly generated graphical user interface that consists of a number of reusable plug-ins. In particular, it includes a preferences panel that lets the user choose which audio backend to use (e.g., JACK or JavaSound, Ambisonics or VBAP) as well as an options panel where the user can set the speed of sound, choose distance cues, and add effects such as reverberation. Then it adds audiovisual content by reading a geometry from a file¹⁹ and launching Csound with the score and orchestra files for Richard Boulanger's *Trapped in Convert*.

The scene is equipped with a number of tools for user interaction, including an ad-hoc tool that pauses and restarts the Csound node. The tool makes no explicit reference to the input device it reacts to; it merely responds to an abstract *panel activation event*. At runtime, the tool system

¹⁷<http://www.csounds.com/>

¹⁸A Windows port of JACK is available, but we have not yet had an opportunity to evaluate it.

¹⁹Our example shows the MATHEON Buddy Bear, whose design was created with jReality.

of jReality will associate this event with a suitable input device. On a desktop system, this will be a double click on the geometry associated with the sound source. In a virtual environment, it may be a wand gesture.

7. OUTLOOK

In spite of its origins in mathematical visualization, jReality has grown to be a general-purpose platform for audiovisual work. We intend to expand it in several new directions.

Future work on jReality will include the generalization of physical simulations to curved geometries, including a physics engine for hyperbolic space and audio capabilities in hyperbolic and elliptic spaces, opening up further possibilities for audiovisual work in an area that has not received nearly enough attention.

Other potential extensions include high-quality audio file rendering backends that are not intended to run in real time, along the lines of the RenderMan backend for graphics.

Finally, we are considering the extension of related work on GPGPU programming[5] to audio processing. Given the computing power of modern graphics cards as well as the tight coupling between audio and graphics in jReality, this approach promises to significantly expand the possibilities for real-time spatialized audio in jReality.

8. ACKNOWLEDGMENTS

We would like to thank all contributors to jReality. We are grateful to Tim Hoffmann and Spencer Topel for many helpful discussions. Michael Gogins provided early insight into the design of the audio backbone and was always ready to help when problems with Csound arose.

9. REFERENCES

- [1] R. W. Furse, "Spatialisation - Stereo and Ambisonic," in *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*, 2000.
- [2] M. Gerzon, "Surround-sound psychoacoustics," *Wireless World*, vol. 80, p. 486, 1974.
- [3] T. Hoffmann and M. Schmies, *Mathematical Software*. Springer, 2006, ch. jReality, jtem, and oorange - a way to do math with computers.
- [4] T. Neumann, C. Fünfzig, and D. Fellner, "TRIPS - A Scalable Spatial Sound Library for OpenSG," in *Proceedings of OpenSG Symposium 2003*, 2003, pp. 57–64.
- [5] U. Pinkall, B. Springborn, and S. Weissmann, "A new doubly discrete analogue of smoke ring flow and the real time simulation of fluid flow," *Journal of Physics A: Mathematical and Theoretical*, vol. 40, no. 42, pp. 12 563–12 576, 2007.
- [6] V. Pulkki, "Spatial sound generation and perception by amplitude panning techniques," Helsinki University of Technology, Tech. Rep., 2001.
- [7] G. Schiemer, S. Ingham, J. Scott, A. Hull, D. Lock, D. Balez, G. Jenkins, I. Burnett, G. Potard, and M. O'Dwyer, "Configurable Hemisphere Environment for Spatialised Sound," 2004.
- [8] J. Sheridan, G. Sood, T. Jacob, H. Gardner, and S. Barrass, "Soundstudio4D - A VR interface for gestural composition of spatial soundscapes," 2004.
- [9] K. van den Doel and D. K. Pai, "Jass: a java audio synthesis system for programmers," in *Proceedings of the 2001 International Conference on Auditory Display*, Helsinki, Finland, 2001.

A. SAMPLE CODE

```
public class Example {
    public static void main(String[] args)
        throws IOException {
        // read geometry from file
        SceneGraphComponent audioComponent =
            Readers.read(Input.getInput("jrs/bear.jrs"));
        // find Csound orchestra and score
        Input input = Input.getInput(
            Example.class.getResource("trapped.csd"));
        // create audio node running Csound
        final AudioSource source =
            new CsoundNode("csound_node", input);
        source.start();
        // attach audio node to geometry
        audioComponent.setAudioSource(source);
        // add some tools for user interaction
        ActionTool actionTool =
            new ActionTool("PanelActivation");
        actionTool.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if (source.getState() ==
                        AudioSource.State.RUNNING) {
                        source.pause();
                    } else {
                        source.start();
                    }
                }
            });
        audioComponent.addTool(actionTool);
        audioComponent.addTool(new DraggingTool());
        // launch viewer and attach content
        JRViewer v = new JRViewer();
        v.addBasicUI();
        v.addAudioSupport();
        v.addVRSupport();
        v.addContentSupport(ContentType.TerrainAligned);
        v.setContent(audioComponent);
        v.startup();
    }
}
```