Guide to SoundSystemJPCT

Overview

SoundSystemJPCT overrides the core SoundSystem class, and is designed to make adding 3D sound to any jPCT project easy. It provides a number of additional methods for things like binding Listener to Camera and Sources to Object3Ds, and using SimpleVector parameters. The jPCT engine utilizes LWJGL for "hardware mode" rendering, and comparatively, SoundSystemJPCT utilizes LWGJL for "hardware mode" sound. The jPCT engine has a pure-Java software rendering mode, and likewise SoundSystemJPCT has a pure-Java option as well.

I will begin by explaining some of the terms used in this guide. If you've already read my previous guide, "3D Sound with SoundSystem", then these terms will look familiar:

- A sample is the sound contained in an audio file.
- A buffer stores the sound bytes for a sample. It may contain the entire sound, or a smaller chunk of data.
- A source is a point in 3D space which emits a sound. A source is not the sample itself, but rather the location where the sample is played.
- A point-source changes as its position in 3D space changes. Close point-sources are louder than those which are further-away. Point-sources to the right or left play louder out of the respective speaker.
- An ambient-source plays at a constant volume regardless of its position in 3D space. Ambient-sources are used for things like background music.
- A streaming source plays a sample in small pieces rather than all at once. Streaming sources are useful for really long samples. A streaming source can be either an ambient-source or point-source.
- Stream-buffers contain small sequential chunks of a sample which are queued to play on a streaming source.
- A channel is the link to an underlying sound library (such as OpenAL or JavaSound), through which sources are played
- The listener is the point in 3D space where sources are heard from. The listener is able to move around in 3D space, and the orientation can be changed by defining the look-at direction and the up-direction.
- Attenuation is how a source's volume fades with distance.
- Rolloff attenuation is a realistic attenuation model, which uses a rolloff-factor to determine how quickly a source's volume fades. A higher rolloff-factor causes a source to fade more rapidly, and a source with zero attenuation never fades (i.e. it is an ambient-source).
- Linear attenuation is less realistic than rolloff-attenuation, but it allows the user to specify a maximum fade-distance, beyond which a source's volume is zero.

The SoundSystemJPCT library and all of the examples used in this guide can be downloaded for free at http://www.paulscode.com. There are also forums there for asking questions, leaving comments, or making suggestions.

At various points in this guide, I will mention my previous guide “3D Sound with SoundSystem”. It is helpful for understanding the core SoundSystem library. I will try to avoid getting into the nuts and bolts of the core library in this guide, so I recommend reading the other guide as well for reference. It is available from the paulscode webside.

Getting Started

A sound file is needed for the example application at the end of this chapter (explosion.wav is used in the code).

Let's begin by learning how to start up and shut down the sound system and how to play sounds. Before we actually connect with jPCT, I'll cover the basics first.

You will of course need SoundSystemJPCT, which can be downloaded from http://www.paulscode.com. Additionally, you will need the jPCT library, which can be downloaded from http://www.jpct.net, and the LWJGL library, which can be downloaded from http://www.lwjgl.org. You will also need a sound file to play. SoundSystemJPCT supports .wav, .ogg (J-Ogg codec included), and .mid formats (NOTE: MIDI works differently than other audio formats – see "3D Sound with SoundSystem" for more information). We are using a file called "explosion.wav" in this example.

By default, your sound file must be compiled into the application JAR in a package named "Sounds", so be sure to place "explosion.wav" there.

First, import the library at the top of your program:

```
import paulscode.sound.SoundSystemJPCT;
```

Starting up the sound system is simple – just create a SoundSystemJPCT object: **Important Note:** There may only be one SoundSystemJPCT instance loaded at a time in your program.

```
SoundSystemJPCT soundSystem = new SoundSystemJPCT();
```

SoundSystemJPCT will automatically try to load LWJGL OpenAL "hardware mode", and if that fails, it will switch to JavaSound "software mode". It is also possible to manually check for compatibility, choose which mode to start in, or add support for additional sound library plug-ins (reference "3D Sound with SoundSystem" for more information about plug-ins and compatibility checking).

At this point, there are several methods you could use for creating a source, but the 'quickPlay' method is probably the one you will use most often. This method automatically creates and begins playing a source. It has numerous variations allowing you to specify as much or as little information as you like. For any parameters not specified, SoundSystemJPCT will automatically use default values. For more information about all the parameters the quickPlay method can take, see "3D Sound with SoundSystem". Let's play an explosion (using "false" for the second parameter means the explosion will only play once, rather than looping over and over):

```
soundSystem.quickPlay( "explosion.wav", false );
```

The last thing you must do is shut down the sound system at the bottom of your program:

```
soundSystem.cleanup();
```

This is very important, because it shuts down whatever library plug-in is active, and frees any resources it is using. Failure to call the cleanup() method when using OpenAL may cause rogue audio artifacts to play continuously on some audio hardware, even after an application has been shut down, forcing the user to reboot. It has even been known to cause some machines to crash, so don't forget to call cleanup() when you are done with the sound system!

Example Application #1

(Creates the SoundSystemJPCT, plays a sound, then shuts down)

```
import paulscode.sound.SoundSystemJPCT;

/**
 *  Creates the SoundSystemJPCT, plays a sound, then shuts down.
 */
public class Example_1
{
    public static void main( String[] args )
    {
        new Example_1();
    }

    public Example_1()
    {
        // Instantiate the SoundSystem:
        SoundSystemJPCT soundSystem = new SoundSystemJPCT();

        // Play an explosion:
        soundSystem.quickPlay( "explosion.wav", false );

        // Wait for 3 seconds:
        sleep( 3 );

        // Shut down:
        soundSystem.cleanup();
    }

    public void sleep( long seconds )
    {
        try
        {
            Thread.sleep( 1000 * seconds );
        }
        catch( InterruptedException e )
        {}
    }
}
```

One more thing to note: There are many default settings in SoundSystemJPCT that can be changed using the SoundSystemConfig class. For example, SoundSystemJPCT by default will look for sound files in the "Sounds/" package. To change this, you can use the setSoundFilesPackage method:

```
SoundSystemConfig.setSoundFilesPackage( "Package_Name/" );
```

If you want have your sound files in multiple packages, then simply call setSoundFilesPackage with an empty string parameter:

```
SoundSystemConfig.setSoundFilesPackage( "" );
```

Throughout this guide, we will only use SoundSystem's built-in default settings, which should be sufficient for most projects. To learn more about SoundSystemConfig and default settings, see "3D Sound with SoundSystem".

Background Music and 3D Sound

A monotone sound effect file and a stereo** music file are needed for the example application at the end of this chapter (explosion.wav and beats.ogg are used in the code).*

The next step is learning how to play background music and sound effects in 3D.

For background music, use the "backgroundMusic" method:

```
soundSystem.backgroundMusic( "beats.ogg" );
```

This creates a priority, looping, ambient, streaming source and begins playing it. If you want to be able to do things to the background music besides just playing it, you will need to give it a source name, using this instead:

```
soundSystem.backgroundMusic( "Cool Music", "beats.ogg" );
```

Then you can use the methods for stop, play, pause, and rewind:

```
soundSystem.stop( "Cool Music" );
```

If you want to create a source for the background music, but not start playing it immediately, use the newStreamingSource method:

```
soundSystem.newStreamingSource( true, "Cool Music", "beats.ogg", true,
                                SoundSystemConfig.ATTENUATION_NONE );
```

We used "true" for the first parameter, so the music is a "priority source" (i.e. it won't be overwritten if too many other streaming sources happen to be playing simultaneously). Using "true" for the fourth parameter means the background music will loop, and using "ATTENUATION_NONE" for the fifth parameter makes the background music ambient.

Playing sounds in 3D is quite simple. The following plays an explosion to the right:

```
soundSystem.quickPlay( "explosion.wav", false, new SimpleVector( 20, 0, 0 ) );
```

The second parameter "false" means the explosion will play once and stop.

Important Note: *Only monotone sound files will work for point sources. If a stereo file is used, the 3D effects will either sound strange (in JavaSound) or simply not work at all (in OpenAL). Stereo sound effects can be easily converted into monotone using a free converter, such as Audacity. Of course, this limitation does not apply to ambient sources.

Important Note #2: **You should only use stereo sound files for ambient sources. This is a limitation of OpenAL – it will apply 3D panning between left and right speakers for monotone files and not for stereo files.

So the moral of the story is: use monotone for sound effects and stereo for music.

When you use quickPlay, a temporary source is created, and it is then removed after it finishes playing. This is generally how you will play sound effects, but if you want a source to stick around, then use the newSource method instead:

```
soundSystem.newSource( "Source 1", "explosion.wav", false );
soundSystem.play( "Source 1" );
```

Example Application #2

(Plays background music and explosions in 3D)

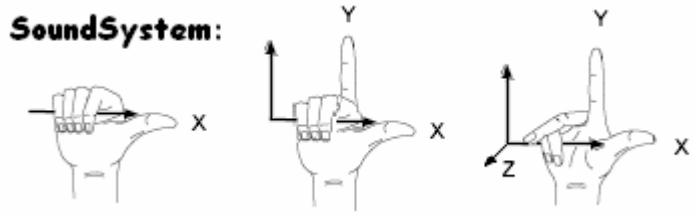
```
import paulscode.sound.SoundSystemJPCT;
// From the jPCT library, http://www.jpct.net
import com.threed.jpct.SimpleVector;

/**
 *  Plays background music and explosions in 3D.
 **/
public class Example_2
{
    public static void main( String[] args )
    {
        new Example_2();
    }

    public Example_2()
    {
        // Instantiate the SoundSystem:
        SoundSystemJPCT soundSystem = new SoundSystemJPCT();

        // play some background music:
        soundSystem.backgroundMusic( "Cool Music", "beats.ogg" );

        // wait a bit before playing the explosions:
        sleep( 2000 );
        // play 15 explosions, right and left:
        for( int x = 0; x < 15; x++ )
        {
            // If x is divisible by 2, play to the right:
            if( x % 2 == 0 )
                soundSystem.quickPlay( "explosion.wav", false,
                                       new SimpleVector( 20, 0, 0 ) );
            // Otherwise play to the left:
            else
                soundSystem.quickPlay( "explosion.wav", false,
                                       new SimpleVector( -20, 0, 0 ) );
            // wait a bit so the explosions don't all start at once
            sleep( 125 );
        }
        // Wait a few seconds:
        sleep( 10000 );

        // Shut down:
        soundSystem.cleanup();
    }

    public void sleep( long miliseconds )
    {
        try
        {
            Thread.sleep( miliseconds );
        }
        catch( InterruptedException e )
        {}
    }
}
```

Interfacing with jPCT

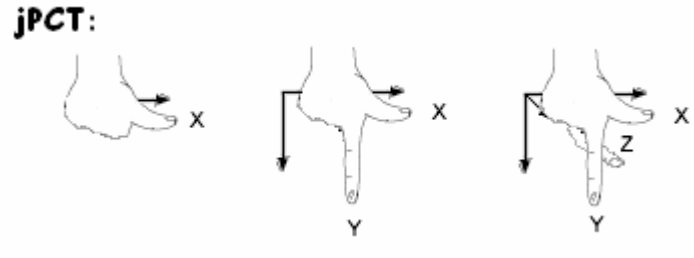
A sound file is needed for the example application at the end of this chapter (explosion.wav is used in the code).

So far we've learned how to start up the sound system, play music, and play sound effects in 3D. The next step is to hook the sound system up to an actual jPCT project.

If you happened to read "3D Sound with SoundSystem", then you might have noticed that the core SoundSystem library uses the right-handed Cartesian 3D coordinate system commonly seen in geometry textbooks and the like:



The jPCT engine also uses a right-handed coordinate system, however it is "upside down" compared to SoundSystem's coordinate system. This is probably due to the fact that in 2D computer graphics, (0, 0) is traditionally the top-left corner of the screen, and +Z is more easily visualized as representing "forward":



You might think this could pose a serious problem when trying to link SoundSystem with jPCT, but don't worry. The SoundSystemJPCT class automatically assumes that any SimpleVector arguments you give it are in the jPCT coordinate system, so it converts the numbers for you before passing them on to the core library. Thus, you should never have to worry about converting between coordinate systems.

Just as jPCT has a "camera" which views the world, SoundSystemJPCT has a "listener" which listens to the world. Since in most cases the "camera" and "listener" are going to be the same, there is a convenient method for linking the two, called "bindListener":

```
soundSystem.bindListener( world.getCamera() );
```

The listener can be “released” from the camera by simply passing null to the bindListener method:

```
soundSystem.bindListener( null );
```

It is important to note that SoundSystemJPCT must be “synced” with jPCT in order for the listener to follow the camera. This is done with the “tick” method:

```
soundSystem.tick();
```

The normal place to stick the call to “tick” is inside your main game loop. Keep in mind that although most of SoundSystemJPCT is thread safe, jPCT is not, so to prevent “bad things” from happening, be sure to either run the “tick” method on the same thread as jPCT, or otherwise synchronize calls to “tick”.

If you find that SoundSystemJPCT is hogging too much of the CPU, you can probably get away with skipping frames (by calling “tick” every other frame or every third frame, for example). The ears are a lot more forgiving than the eyes, so players are much more likely to see the jerky movement caused by a reduced frame rate than they are to hear sound effects that are not moving perfectly smooth.

The other way to hook SoundSystemJPCT up to jPCT is by binding sources to Object3Ds. This works pretty much the same way as binding the listener to the camera:

```
soundSystem.bindSource( "sourcename", someObject3D );
```

The above line makes it so that when “sourcename” is playing, it will “follow” the Object3D around. For example, a buzz sound can fly around with a mosquito without you having to worry about changing the source’s position to match the mosquito.

For obvious reasons, a source can only be bound to one Object3D at a time. Additional calls to “bindSource” using the same source name will simply move the source over to the new Object3D. You can, however, bind as many sources as you like to each Object3D.

To “release” a source from its Object3D, call the “releaseSource” method:

```
soundSystem.releaseSource( "sourcename" );
```

There is also a “releaseAllSources” method which will release any sources that are bound to an Object3D:

```
soundSystem.releaseAllSources( someObject3D );
```

Important Note: When binding sources to Object3Ds, you should always call releaseAllSources before discarding an Object3D, so that references to it are removed, and the Java garbage collector can free it from memory!

Just as when using the “bindListener” method, using “bindSource” requires calls to “tick” to keep SoundSystemJPCT in sync.

The “quickPlay” method is probably what you will be using the most often for non-looping sound effects (like explosions, lasers, etc). Because of this, there are several variations of the “quickPlay” method that take an Object3D as a parameter instead of a SimpleVector. Using these methods will automatically bind the new temporary source to the Object3D:

```
soundSystem.quickPlay( "flame.wav", false, enemyShip );
```

In the above example, the flame sound effect will follow the ship’s movements until it finishes playing, and then it will be automatically removed. This gives you another convenient way to “play it and forget it”.

Example Application #3

(Binds the listener to the camera and a source to an Object3D)

```
// Author: Paul Lamb (http://www.paulscode.com)
import paulscode.sound.SoundSystemJPCT;

// From the jPCT library, http://www.jpct.net
import com.threed.jpct.Camera;
import com.threed.jpct.FrameBuffer;
import com.threed.jpct.IRenderer;
import com.threed.jpct.Object3D;
import com.threed.jpct.Primitives;
import com.threed.jpct.SimpleVector;
import com.threed.jpct.World;
import com.threed.jpct.util.KeyMapper;
import com.threed.jpct.util.KeyState;

import java.awt.BorderLayout;
import java.awt.Canvas;
import java.awt.Graphics;
import java.awt.Insets;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;

/**
 *  Binds the listener to the camera and a source to an Object3D.
 **/
public class Example_3 extends JFrame implements Runnable
{
    SoundSystemJPCT soundSystem;

    // size to make the JFrame:
    private int width = 640;
    private int height = 480;

    // top and left insets for the JFrame:
    private int titleBarHeight = 0;
    private int leftBorderWidth = 0;

    // setting to false ends the main game loop:
    private boolean running = true;

    // used for keeping track of keyboard input:
    private KeyMapper keyMapper = null;

    // booleans indicating when the arrow keys are pressed:
    private boolean upKeyPressed = false;
    private boolean downKeyPressed = false;
    private boolean rightKeyPressed = false;
    private boolean leftKeyPressed = false;

    // speeds to use for turning and moving the camera:
    private float turnStep = 0.01f;
    private float moveStep = 1.0f;

    // set to true when initialization is complete:
    private boolean initialized = false;

    // jPCT stuff:
    private Object3D box;
    private FrameBuffer buffer = null;
    private World world = null;
```

```
    // For jPCT to render on:
    Canvas myCanvas;

    // Used to synchronize multiple threads interfacing with jPCT:
    private final Object jPCTLock = new Object();

    public static void main( String[] args )
    {
        new Example_3();
    }

    public Example_3()
    {
        // set up the JFrame:
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setTitle( "Example 3" );
        pack();
        Insets insets = getInsets();
        titleBarHeight = insets.top - 1;
        leftBorderWidth = insets.left - 1;
        setSize( width + leftBorderWidth + insets.right - 1,
                 height + titleBarHeight + insets.bottom - 1 );
        setResizable( false );
        setLocationRelativeTo( null );
        setVisible( true );

        // initialize jPCT and the sound system:
        init();
    }

    public void init()
    {
        synchronized( jPCTLock )
        {
            world = new World();  // create a new world
            World.setDefaultThread( Thread.currentThread() );

            // Set the overall brightness of the world:
            world.setAmbientLight( 50, 50, 50 );

            // Create a main light-source:
            world.addLight( new SimpleVector( 0, -50, -50 ), 20, 20, 20 );


            // create a new framebuffer and canvas to render on:
            buffer = new FrameBuffer( width, height,
                                      FrameBuffer.SAMPLINGMODE_HARDWARE_ONLY );
            buffer.disableRenderer( IRenderer.RENDERER_SOFTWARE );
            myCanvas = buffer.enableGLCanvasRenderer();

            // Create a box:
            box = Primitives.getBox( 3.0f, 1.0f );
            box.build();

            // add the box our world:
            world.addObject( box );

            // set the camera's position:
            world.getCamera().setPosition( 0, 0, -100 );
        }

        // add the canvas to the JFrame and make it visible:
```

```
        add( myCanvas, BorderLayout.CENTER);
        myCanvas.setVisible( true );

        // receive keyboard input:
        keyMapper = new KeyMapper( this );
        addKeyListener( keyMapper );
        myCanvas.addKeyListener( keyMapper );

        // initialize the sound system:
        soundSystem = new SoundSystemJPCT();

        synchronized( jPCTLock )
        {
            // bind the listener to the camera:
            soundSystem.bindListener( world.getCamera() );

            // bind a new looping source to the box:
            soundSystem.quickPlay( "explosion.wav", true, box );
        }

        // finished initializing:
        initialized = true;

        // start the main game loop
        new Thread(this).start();
    }

    // Main game loop:
    @Override
    public void run()
    {
        while( running )
        {
            // check for keyboard input:
            pollKeyboard();

            // sync the soundsystem:
            soundSystem.tick();

            // move the camera around when the arrow keys are pressed:
            if( upKeyPressed )
            {
                synchronized( jPCTLock )
                {
                    world.getCamera().moveCamera( Camera.CAMERA_MOVEIN,
                                                  moveStep );
                }
            }
            if( downKeyPressed )
            {
                synchronized( jPCTLock )
                {
                    world.getCamera().moveCamera( Camera.CAMERA_MOVEOUT,
                                                  moveStep );
                }
            }
            if( rightKeyPressed )
            {
                synchronized( jPCTLock )
                {
                    world.getCamera().rotateY( -turnStep );
                }
            }
```

```
            if( leftKeyPressed )
            {
                synchronized( jPCTLock )
                {
                    world.getCamera().rotateY( turnStep );
                }
            }

            // tell the JFrame to repaint itself:
            this.repaint();

            // limits the fps:
            try
            {
                Thread.sleep( 20 );
            }
            catch(Exception e){}
        }

        // Important!
        soundSystem.cleanup();
    }

    // Render the scene
    @Override
    public void paint( Graphics g )
    {
        // wait for initialization to complete:
        if( !initialized )
            return;

        buffer.clear();   // erase the previous frame

        // render the world onto the buffer:
        world.renderScene( buffer );
        world.draw( buffer );
        buffer.update();

        // Repaint the canvas:
        buffer.displayGLOnly();
        myCanvas.repaint();
    }

    // Check for keyboard input:
    private void pollKeyboard()
    {
        KeyState state = null;
        do
        {
            state = keyMapper.poll();
            if( state != KeyState.NONE )
            {
                keyAffected( state );
            }
        } while( state != KeyState.NONE );
    }

    // Process keyboard input:
    private void keyAffected( KeyState state )
    {
        int code = state.getKeyCode();
        boolean event = state.getState();
```

```
        switch( code )
        {
            case( KeyEvent.VK_UP ):
            {
                upKeyPressed = event;
                break;
            }
            case( KeyEvent.VK_DOWN ):
            {
                downKeyPressed = event;
                break;
            }
            case( KeyEvent.VK_RIGHT ):
            {
                rightKeyPressed = event;
                break;
            }
            case( KeyEvent.VK_LEFT ):
            {
                leftKeyPressed = event;
                break;
            }
            case( KeyEvent.VK_ESCAPE ):
            {
                running = event;
                break;
            }
        }
    }
}
```

Conclusion

In the last example, you saw how only a couple lines of code were needed for using the sound system (most of the code was for setting up the JFrame and keyboard input). And that is the point – SoundSystemJPCT was designed to make 3D sound simple, so you can spend less time worrying about the sound engine and more time developing the rest of your project.

At this point, you should have all the skills you need to incorporate sound into your jPCT projects. The rest is up to your own creativity. If you have any comments or suggestions about this book or about anything related to the SoundSystem library, feel free to post a thread on the forums at http://www.paulscode.com.

Legal Concerns

I the author do not guarantee, warrant, or make any representations, either expressed or implied, regarding the content of this book. The examples are provided “as is”, and I the author will not be responsible for any damages (physical, financial, or otherwise) caused by their use.

References used in this book:

jPCT
The jPCT API (http://www.jpct.net)

LWJGL
The Lightweight Java Gaming Library (http://www.lwjgl.org)

J-Ogg
The J-Ogg library (http://www.j-ogg.de), copyrighted by Tor-Einar Jarnbjo

The SoundSystem License:

You are free to use this library for any purpose, commercial or otherwise.
You may modify this library or source code, and distribute it any way you like, provided the following conditions are met:
1) You may not falsely claim to be the author of this library or any unmodified portion of it.
2) You may not copyright this library or a modified version of it and then sue me for copyright infringement.
3) If you modify the source code, you must clearly document the changes made before redistributing the modified source code, so other users know it is not the original code.
4) You are not required to give me credit for this library in any derived work, but if you do, you must also mention my website: http://www.paulscode.com
5) I the author will not be responsible for any damages (physical, financial, or otherwise) caused by the use if this library or any part of it.
6) I the author do not guarantee, warrant, or make any representations, either expressed or implied, regarding the use of this library or any part of it.
Author: Paul Lamb
http://www.paulscode.com