Demicron

Reference Manual

Java in WireFusion 5

WireFusion API & SDK

Contents

INTRODUCTION **1**

PART I: THE JAVA OBJECT **2**

- INTRODUCTION 2
  - About this chapter 4
- REQUIREMENTS 5
  - User Requirements 5
  - System Requirements 5
- SHORTCUTS 6
- DEVELOPMENT ENVIRONMENT 7
  - Menu items 7
  - Ports 9
- SYSTEM EVENTS 14
- USING AWT COMPONENTS 15
  - Example: Adding a Pop-up menu 15
  - Exercise: Adding a Text field 19
- RESOURCES 22
  - Java Libraries 22
  - Resource Files 23
  - Exercise: Reading text from a resource file 24
- NOTES 26

PART II: WIREFUSION SDK **27**

- INTRODUCTION 27
  - About this chapter 27
- SETTING UP THE ENVIRONMENT 28
  - Javac and simple text editor 28
  - Eclipse 3.2 29
- BASICS OF CREATING A WOB 31
  - Creating the player class 31
  - Creating the Wob properties class 34
  - Configuring the Wob 39
- HELP PAGES 43
- BUILDING AND INSTALLING THE WOB 44
- FILE DEPENDENCIES 45
- CUSTOM CONFIGURATION DIALOGS 49
- CUSTOM TARGET AREA VIEW 49
- ADD-ON BUILDER 49
  - Serial Number validation 50
  - Splash Screens 51
  - License text 51
  - Building 51

PART III: WIREFUSION 3D API **52**

- INTRODUCTION 52
  - About this chapter 52
- REQUIREMENTS 53
  - User Requirements 53
  - System Requirements 53
- DEFINITIONS 54
  - WireFusion W3F Format 54
  - X3D and VRML Definitions 54
  - WireFusion 3D API Definitions 55
- BASICS 57
  - Get the browser 59
  - Get the scene 59

Get a named node 59
Get a field 59
Read and change a field 60
Listening to field changes 60
Load new textures 60
Load an external 3D file 60
Add and remove a 3D file 61
Replace a 3D file 61
BASIC EXCERCISES 62
Exercise: Adding a 3D file to a 3D scene 62
Exercise: Changing the texture of an object 68
APPENDIX: SUPPORTED NODES AND FIELDS 72
Appearance 72
DirectionalLight 72
Group 73
Material 74
PointLight 75
Shape 76
Texture 76
TextureTransform 76
TimeSensor 77
TouchSensor 77
Transform 78
Viewpoint 79
INDEX 80

Introduction

A powerful Java API is available in WireFusion. It can be used for building your own WireFusion objects, for controlling 3D Scene objects and to dynamically control objects and wires during presentation runtime. There are two ways to access the Java API; through the Java object or through WireFusion objects created using the WireFusion SDK. It is easy to get started using the Java object and its integrated development environment. If you want to use your own development environment instead, or feel restrained by some of the restrictions of the Java object, then you will want to use the WireFusion SDK.

The first two chapters will cover the Java Object and the WireFusion SDK. The last chapter covers the WireFusion 3D API.

Part I: The Java Object

Introduction

The WireFusion Java object allows programmers to easily write and compile their own Java code (programs) directly in WireFusion. Non-programmers can also benefit from the Java object, and the power of the Java language, by obtaining free and ready-made source code from either the Demicron web site or from third parties.

The Java object, found in the Misc category in WireFusion (Figure 1), can interact with the rest of the WireFusion environment through In-ports and Out-ports or by calling other objects directly through the API. One important use of the Java object is to control 3D Scene objects through the 3D API (see next chapter). The integrated Java development environment is easy to learn and will get you started in no time.

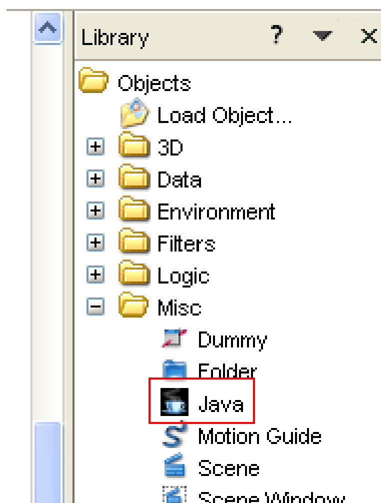

Figure 1: The Misc category, containing the Java object

Java objects can be saved and reused in other projects, or shared with other WireFusion users. After your Java object has been coded, compiled and is ready for use, you can save it using the Java object's local menu or add it to the Library as a Favorite (Figure 2). If you want to share your Java object with other WireFusion users, but not reveal the source code, then you can password protect the object when distributing it as a Favorites library.

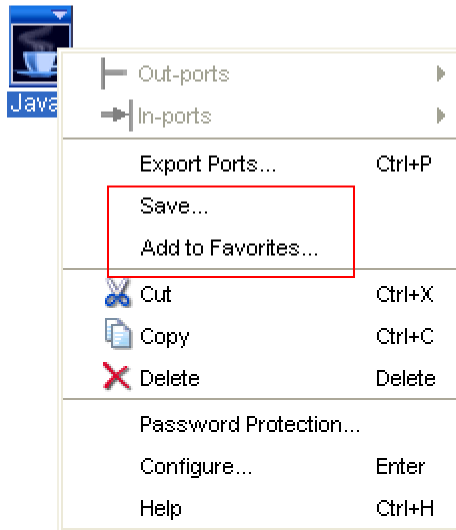

Figure 2: Java object's local menu

Some application areas of the Java object:

- Communicate with the 3D Scene object using the 3D API
- Create Widgets (user interface components)
- Create advanced functions (logic)
- Evaluate complex mathematical expressions
- Communicate with databases and web services
- Create image processing filters
- Read and process external data, like xml files

TIP: If you want to create a new Wob using the WireFusion SDK, then it can be a good idea to first create a prototype using the Java object. The code from the Java object can then easily be converted later to a "native" WireFusion object using the WireFusion SDK.

You can find a full API documentation at this URL:

http://www.demicron.com/wirefusion/v5/docs/api/index.html

NOTE: Many of the classes in this documentation are only meant to be used with the WireFusion SDK, not the Java object. The classes of most interest for Java object developers are located in the `com.wirefusion.player` and `wobs.java.player` packages.

If you want your presentation to be Java 1.1 compatible, then you must restrict your code to only use the Java 1.1 API and avoid non Java 1.1 compatible syntax. The Java 1.1 API documentation can be downloaded from here:

http://java.sun.com/products/archive/jdk/1.1/index.html

IMPORTANT: The API has undergone many changes since WireFusion 4 and 3. In projects created using a WireFusion version prior to 5.0, the Java source will be automatically converted. If this conversion fails, an error message will be shown and a manual conversion may be necessary. Please contact Demicron if this occur and if possible include the old Java source code that could not be converted.

Please send comments and feedback regarding this manual or the software to contact@demicron.com

About this chapter

In this chapter we will introduce the WireFusion Java object and explain how to write and compile Java code from inside WireFusion. We will explain the basics through some examples and exercises.

Requirements

User Requirements

To take full advantage of the Java object in WireFusion you should be familiar with the Java programming language. Non-programmers can also take advantage of the Java object by re-using already programmed Java objects, or by copying and pasting Java source code created by other users.

If you've never used WireFusion before, then it is highly recommended that you work through the tutorial ***Getting Started, Volume I***.

System Requirements

You need to have WireFusion Professional edition, WireFusion Enterprise edition or WireFusion Educational edition version 5.0.2 or higher installed on your computer.

Shortcuts

Edit	**PC/Linux**	**Mac**
Cut	CTRL + X	Command + X
Copy	CTRL + C	Command + C
Paste	CTRL + V	Command + V
Find…	CTRL + F	Command + F
Replace…	CTRL + R	Command + R
Undo	CTRL + Z	Command + Z
Redo	CTRL + Y	Command + Y
Preferences	CTRL + P	Command + P

Source	**PC/Linux**	**Mac**
Verify Source	CTRL + F7	Command + F7

Tools	**PC/Linux**	**Mac**
Insert In-port code	CTRL + I	Command + I
Insert Out-port code	CTRL + O	Command + O

Development Environment

To program a Java object in WireFusion, drag in the Java object, found in the Misc category. When you drop the Java object, you will be asked if you want the object to have a Target Area. If you want the object to display graphics in the presentation, and/or listen to Mouse or Keyboard events, then click “Yes”, otherwise “No”. The object dialog window will now be opened and you will be presented with an editable Java source code body, which will become your main class (Figure 3). If you specified that the object should have a Target Area, then an empty `paint()` method will also be included in the initial source code body. When you have created your program, close the Java object dialog by clicking “OK”. The source is compiled and in-ports and out-ports, defined by your source code, are automatically generated.

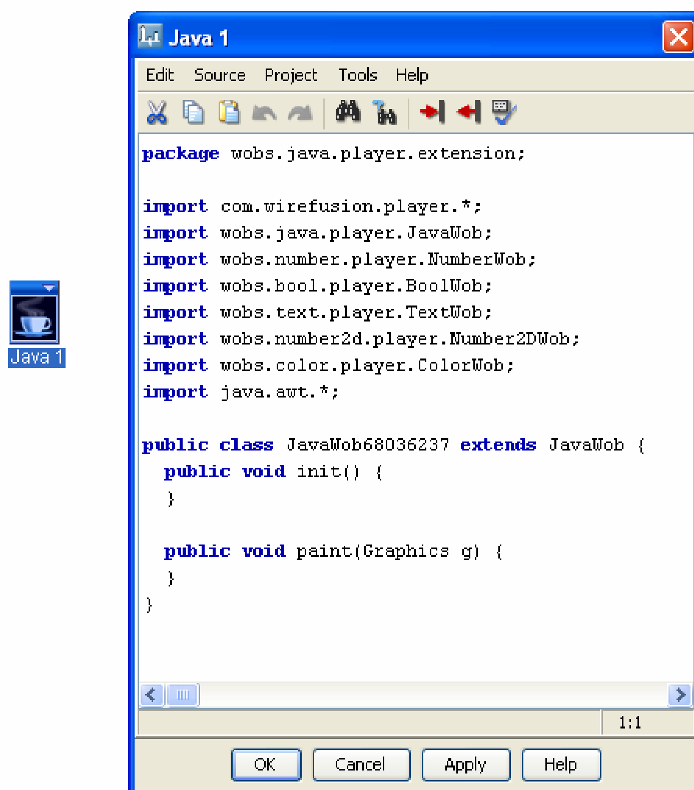

Figure 3: The Java object’s dialog window

Menu items

The Java development environment is very basic and easy to use. The different menu options are described below.

Edit

Cut

Cuts text and places it on the clipboard.

Copy
Copies text and places it on the clipboard.

Paste
Pastes text from the clipboard.

Find…
Finds text in the code.

Replace…
Finds and replaces text in the code.

Undo
Undo the last text addition/deletion.

Redo
Redo the last text addition/deletion.

Preferences
Opens the preferences dialog.

Source

Verify Source
Checks for errors in the source code.

Tools

Insert In-port code
Auto-generates and inserts code for creation of in-ports.

Insert Out-port code
Auto-generates and inserts code for creation of out-ports.

Ports

In-ports

To add an in-port, you must add a Java method, formatted in a certain way, to the main class. The argument of the method decides what argument the in-port should have, and the name of the in-port is specified by the name of the method. In the method body you add the code that you would like to be executed when an event is sent to the in-port. Below you can see the code format to use for the six different in-port types supported.

```
public void inport_<inport-name>()
```

Defines an in-port named <inport-name> with no argument (pulse).

```
public void inport_<inport-name>(double d)
```

Defines an in-port named <inport-name> with a Number as argument.

```
public void inport_<inport-name>(boolean d)
```

Defines an in-port named <inport-name> with a Boolean as argument.

```
public void inport_<inport-name>(String  d)
```

Defines an in-port named <inport-name> with a Text as argument.

```
public void inport_<inport-name>(double x, double y)
```

Adds an in-port named <inport-name> with a 2D Number as argument.

```
public void inport_<inport-name>(int color)
```

Adds an in-port named <in-port-name> with a Color as argument. The int value specifies the color and has the format AARRGGBB (AA is currently not used).

NOTE: If you want to have spaces in the port name, use underscores ('_'). The underscore will be presented as a space in the port menu that will be generated once the code is compiled.

NOTE: If you want to place a port name under a sub menu, then use the following composite port name: <sub menu name>_submenu_<port name>. An example is `Sound_submenu_Start`, which would generate a sub menu named Sound with a port named Start.

Example: Creating an In-port

```
// Entering the following code into a Java object without
// Target Area results in an object with an in-port named
// "Print", that has a Text object as argument.
// Events to this in-port causes the Java object to print
// the Text value to the standard Java output (Java Console).
```

```
package wobs.java.player.extension;

public class JavaWob123 extends wobs.java.player.JavaWob {
      public void inport_Print(String arg) {
         System.out.println(arg);
      }

      public void init() {
      }
}
```

NOTE: The class name of the above code will not be accepted by the Java object when you compile the code. When the Java object complains about the class name, choose “Yes”, and the class name will be automatically corrected.

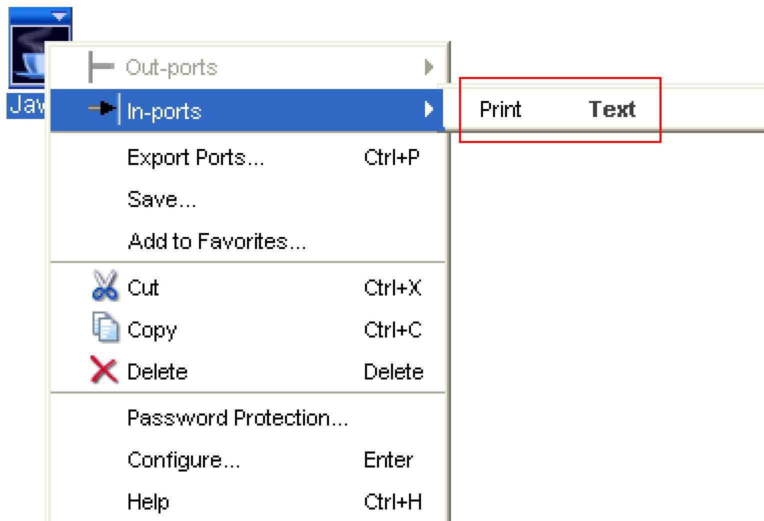

Figure 4: The new in-port Print

Out-ports

You can also *send* events from the Java object, through out-ports, by calling one of the following methods from your main class:

```
public void sendPulse(String portName)
```

Sends a pulse event through a port named portName.

```
public void sendNumber(String portName, double argument)
```

Sends a Number event through a port named portName.

```
public void sendBoolean(String portName, boolean argument)
```
Sends a Boolean event through a port named portName.

```
public void sendText(String portName, String argument)
```
Sends a Text event through a port named portName.

```
public void send2DNumber(String portName, double x, double y)
```
Sends a 2D Number event through a port named portName.

```
public void sendColor(String portName, int color)
```
Sends a Color event through a port named portName. The color argument should have the format AARRGGBB, where each letter represents a byte in the color `int` value. R is red, G is green and B is blue (AA currently not used).

Example sendColor:

```
sendColor("My Port", 0xFF0000); // Sends the color blue through "My Port"
```

When exiting the development environment, your Java code is automatically analyzed and checked for calls to these methods, and the appropriate out-ports are automatically created.

IMPORTANT: The portName argument must be an explicit String (i.e. not a String reference), otherwise the code analyzer will not be able to find the port name.

Example: Create an out-port

```
// A Java object with this code can receive a Number value,
// calculate the sin value of the incoming Number
// value, and then send the result through an out-port.

package wobs.java.player.extension;

public class JavaWob123 extends wobs.java.player.JavaWob {
	public void inport_Calculate_Sine(double arg) {
	   sendNumber("Result", java.lang.Math.sin(arg));
	}
}
```

NOTE: The class name of the above code will not be accepted by the Java object when you compile the code. When the Java object complains about the class name, choose "Yes", and the class name will be automatically corrected

---

NOTE: You can define sub menus and spaces for out-ports in the same way you do for in-ports.

Auto Generate Ports

An alternative to manually entering code for in-ports and out-ports is to use the menu options "Insert In-port code" and "Insert Out-port code", which auto-generates code.

Insert In-port code

"Insert In-port code" allows you to quickly insert the code, at the current cursor position, needed to create an in-port. When chosen, either from the Tools menu or from the toolbar (Figure 5), the In-port code generator dialog is opened (Figure 6).

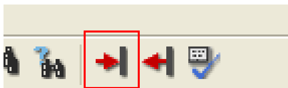

Figure 5: The Insert In-port code button

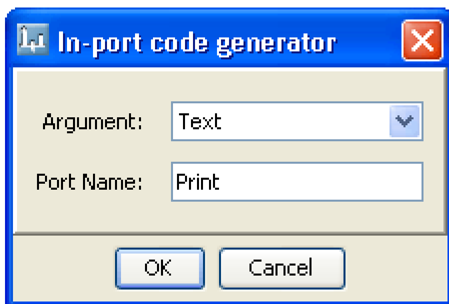

Figure 6: The In-port code generator dialog

Argument

Choose which type of argument the in-port shall have: No Argument (pulse), Number, Boolean, Text, 2D Number or Color.

Port Name

Choose a name for the in-port.

Insert Out-port code

Insert Out-port code allows you to quickly insert the code defining an out-port at the current cursor position. Open the "Insert Out-port code" dialog, either from the Tools menu or from the tool bar (Figure 7), the Out-port code generator dialog is opened (Figure 8).

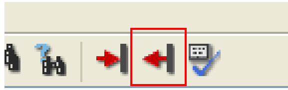

Figure 7: The Insert Out-port code button

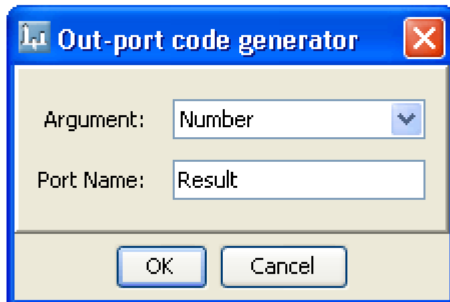

Figure 8: The Out-port code generator dialog

Argument

Choose which type of argument the out-port shall have: No Argument (pulse), Number, Boolean, Text, 2D Number or Color.

Port Name

Choose a name for the out-port.

System Events

Following is a description of how you can code your Wob to react to certain system/player events, like when a new frame has been shown in the player or when the presentation starts. Check the API documentation for the class com.wirefusion.player.Wob for more details.

Player events:

A number of events generated by the player can be listened to by enabling the PlayerEvent type in your Wob (see API documentation for com.wirefusion.player.Wob):

```
enableEvents(PlayerConstants.PLAYER_EVENT_MASK);
```

When you have enabled Player events, they will be sent to the `processPlayerEvent()` method of your JavaWob.

AWT events:

You can listen to AWT events by enabling them in a similar way to as you do in the class java.awt.Component. For example, to listen to key events and mouse motion events, enable them like this:

```
enableEvents(AWTEvent.KEY_EVENT_MASK | AWTEvent.MOUSE_MOTION_EVENT_MASK);
```

You must also override the methods `processKeyEvent()` and `processMouseMotionEvent()` to listen to the events.

Using AWT Components

You can add AWT (Abstract Windowing Toolkit) components, like text fields and pop-up-menus, to WireFusion presentations using the Java object. This can be useful if there is no corresponding WireFusion component available. AWT components do not work as normal visual WireFusion components. For example, they will not be visible in the Layers view, and, they are not included in the WireFusion layer hierarchy. They are always placed on top of the presentation. This means that you cannot have, for example, a lens effect on an AWT button; the AWT button will always be placed above the lens effect. Lightweight components, like Swing components, are not supported, but they can often easily be converted to a Java object since the Java object mimics java.awt.Component.

Example: Adding a Pop-up menu

In this example we will add a WireFusion button that will show an AWT pop-up menu when you click the button. We will add two items to the pop-up menu. The first item is a standard menu item that, which when selected will show a label named "Item has been selected". The other menu item is a checkbox that will show/hide a label named "Checkbox checked".

Step 1

Insert a Button and two Label objects into an empty project.

In the Button dialog:

- Set the label to "Show pop-up menu"

In the Label dialogs:

- For 'Label 1', set the label to "Item has been selected".
- For 'Label 2', set the label to "Checkbox checked".

Step 2

Deactivate both Label objects at the presentation startup by deselecting their Activated checkboxes, found in the Layers view (Figure 9).

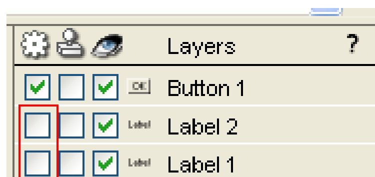

Figure 9: Deactivating the Label objects in the Layers view

Step 3

Drag in a Java object without Target Area and enter the following code.

```
package wobs.java.player.extension;

import com.wirefusion.player.*;
import wobs.number.player.NumberWob;
import wobs.bool.player.BoolWob;
import wobs.text.player.TextWob;
import wobs.number2d.player.Number2DWob;
import wobs.color.player.ColorWob;
import java.awt.*;
import java.awt.event.*;

public  class  JavaWob123  extends  wobs.java.player.JavaWob  implements
ActionListener, ItemListener {

  PopupMenu popupMenu = new PopupMenu();
  MenuItem menuItem = new MenuItem("Standard item");
  CheckboxMenuItem checkboxItem = new CheckboxMenuItem("Checkbox item");

  public void inport_popup() {
    popupMenu.show(getCore().getDisplay().getAwtContainer(),
      getCore().getMousePosition().x, getCore().getMousePosition().y);
  }

  public void actionPerformed(ActionEvent ev) {
    if (ev.getSource()==menuItem) {
      sendPulse("Standard item selected");
    }
  }

  public void itemStateChanged(ItemEvent ev) {
    if (ev.getSource()==checkboxItem) {
      if (checkboxItem.getState())
        sendPulse("Checkbox checked");
      else
        sendPulse("Checkbox unchecked");
    }
  }

  public void init() {
    popupMenu.add(menuItem);
    menuItem.addActionListener(this);
    popupMenu.add(checkboxItem);
    checkboxItem.addItemListener(this);
    getCore().getDisplay().getAwtContainer().add(popupMenu);
  }
}
```

NOTE: The class name of the above code will not be accepted by the Java object when you compile the code. When the Java object complains about the class name, choose “Yes”, and the class name will be automatically corrected

Step 4

Close the Java object and make the following connections:

Connect:

- *'Button 1' > Out-ports > Button Clicked*
  to
  'Java 1' > In-ports > popup

- *'Java 1' > Out-ports > Standard item selected*
  to
  'Label 1', In-ports > Activate

- *'Java 1' > Out-ports > Checkbox checked*
  to
  'Label 2', In-ports > Activate

- *'Java 1' > Out-ports > Checkbox unchecked*
  to
  'Label 2' > In-ports > Deactivate

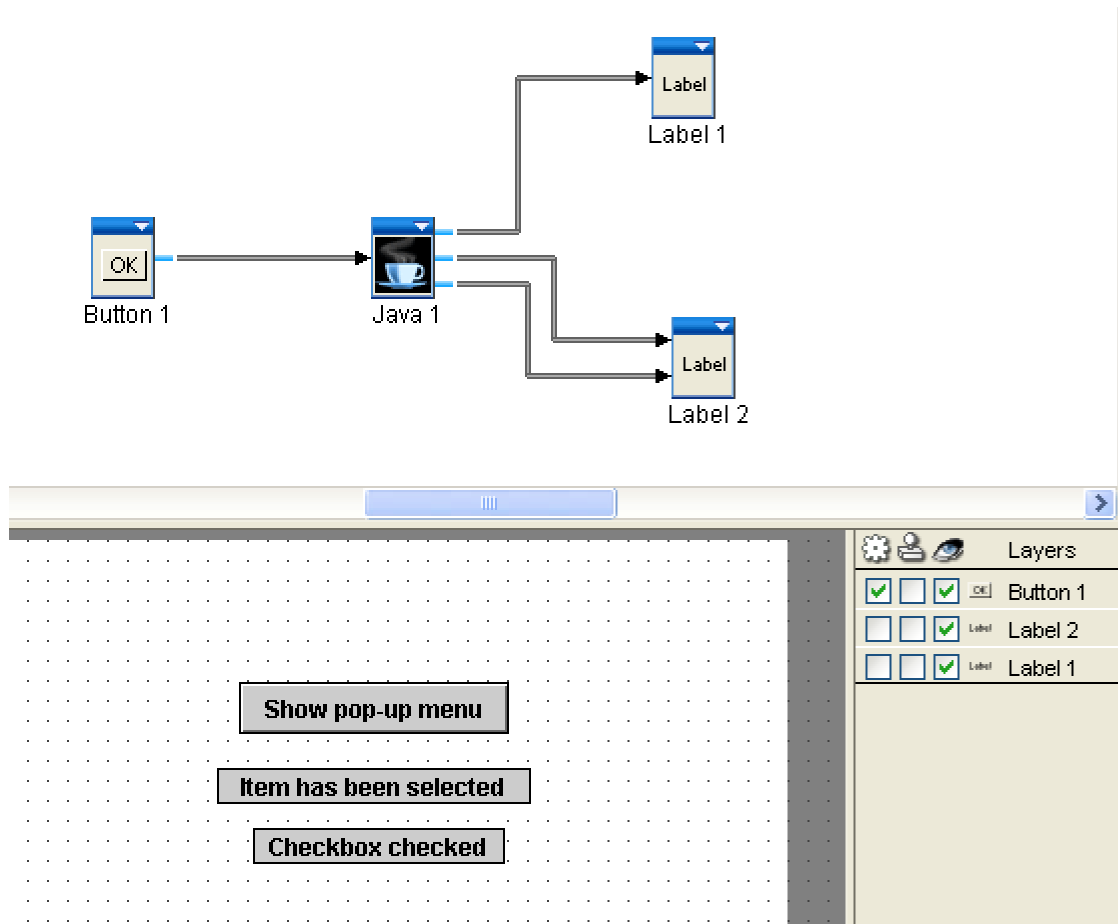

Figure 10: Objects and connections done for the pop-up menu example

Step 5

Now you can view the presentation and see the result of using the pop-up menu. Press F7 to test your presentation (Figure 11). Since the popup menu locks the system event thread (which is the thread calling the inport_popup() method and also the thread running the presentation), the presentation will be frozen while the popup menu is displayed.

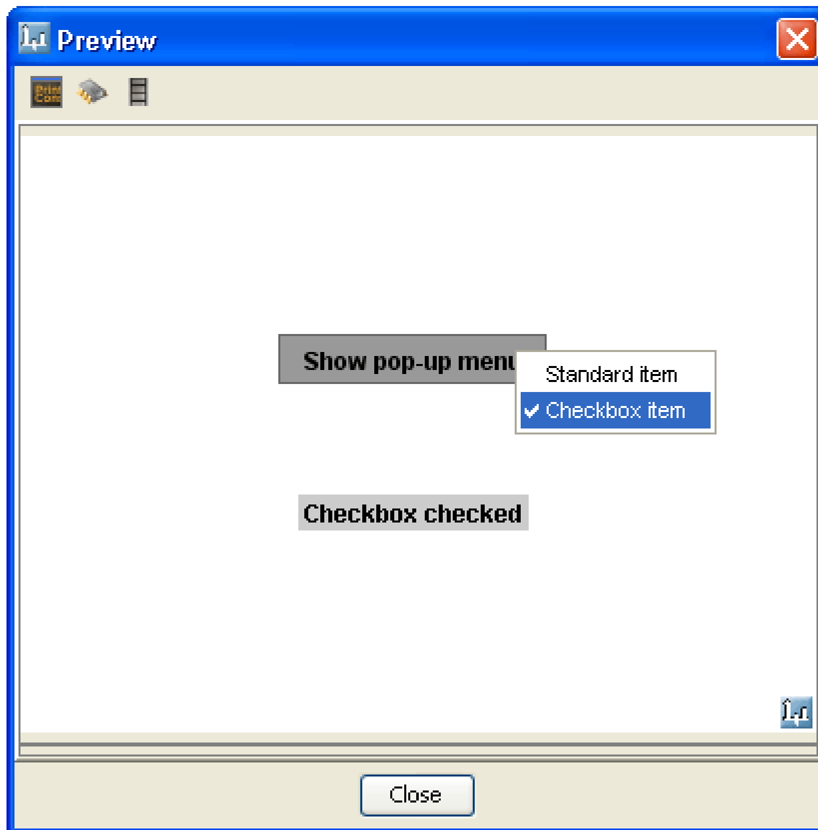

Figure 11: Preview of AWT popup menu example

Exercise: Adding a Text field

In this example a text field will be added to the presentation, and when the Enter key is pressed in the text field, the text will be shown in the status bar.

Step 1

Insert a Java object without Target Area into an empty project and enter the following code.

```
package wobs.java.player.extension;

import com.wirefusion.player.*;
import wobs.number.player.NumberWob;
import wobs.bool.player.BoolWob;
import wobs.text.player.TextWob;
import wobs.number2d.player.Number2DWob;
import wobs.color.player.ColorWob;
import java.awt.*;
import java.awt.event.*;

public class JavaWob123 extends wobs.java.player.JavaWob implements
ActionListener {
  TextField textField = new TextField("Enter a text and press enter");

  public void init() {
```

```
    getCore().getDisplay().getAwtContainer().setLayout(null); // null layout
    getCore().getDisplay().getAwtContainer().add(textField);
    // Use setBounds since no layout manager is use
    textField.setBounds(20,50,200,
    textField.getPreferredSize().height);
    textField.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ev) {
    sendText("Text entered", textField.getText());
  }
}
```

NOTE: The class name of the above code will not be accepted by the Java object when you compile the code. When the Java object complains about the class name, choose “Yes”, and the class name will be automatically corrected

---

Step 2

Insert a System object into the project and make the following connection.

Connect:

'Java 1' > *Out-port* > *Text entered*
to
'System 1' > *In-ports* > *Set Status Bar Text*

Step 3

Press F7 to test your presentation (Figure 12).

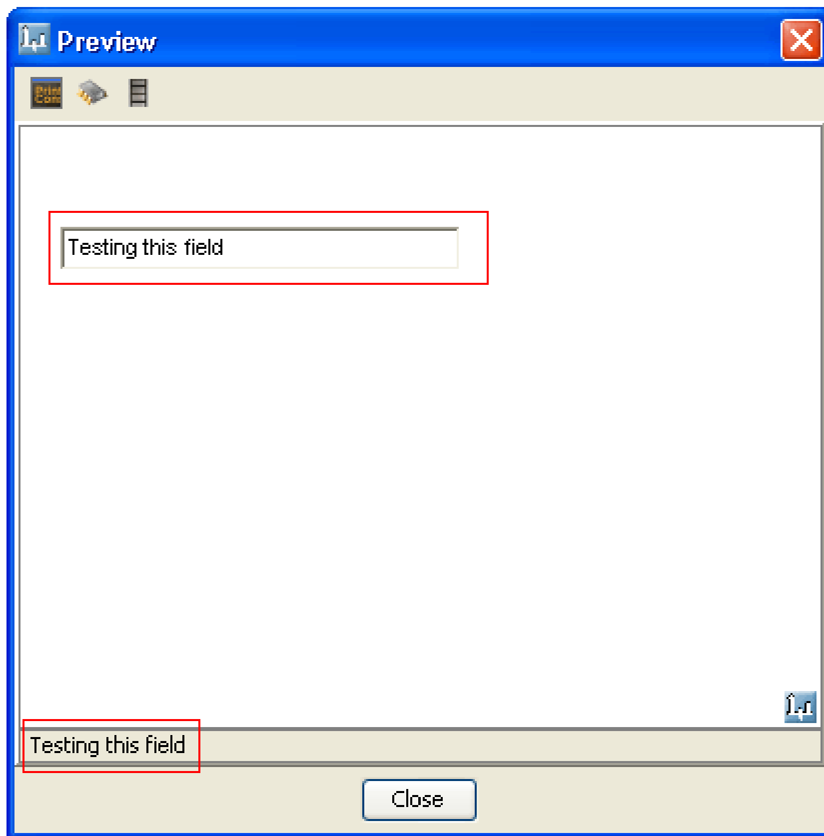

Figure 12: Printing a text in the status bar

Resources

In the Java Resources dialog (Figure 13), which is opened with Project > Resources… in the Java object dialog, you can specify external resource files and Java libraries. These resources and libraries can then be used by the Java object.

Java Libraries

To specify a Java library, select the Java Libraries panel in the Resources dialog, and then add your Java Archive file (*.jar*). You can now use the classes in your archive in your Java object class.

Examples of useful libraries:

- Communicate with a MySQL server using the MySQL Connector library, available from *http://www.mysql.com/*
- Nano XML, a very compact XML parser, available from *http://nanoxml.cyberelf.be/*

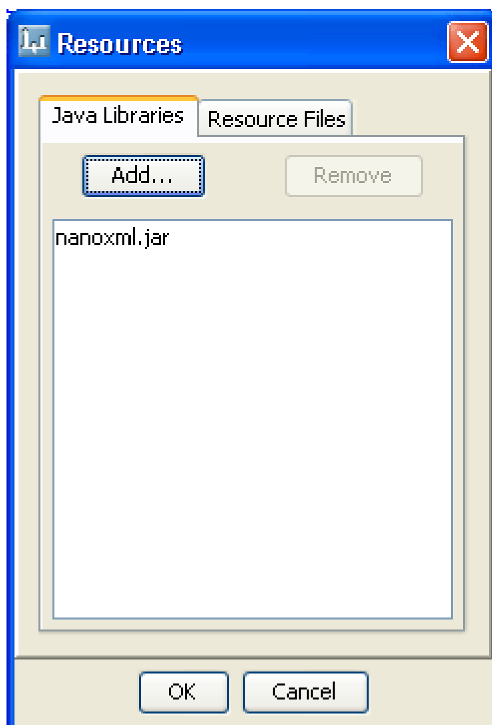

Figure 13: NanoXML library added

Creating your own Java libraries

It is easy to create your own class libraries that can be imported into the Java object. You can develop the libraries in your favorite Java development.

The class files must be packaged in a JAR file before they can be imported into the Java object. For example, if you want to create the Java library from the class files in a package

called `mypackage`, located in the folder *c:\mypackage*, then execute the following command from a command shell:

> *jar cvf MyClasses.jar mypackage*.class*

MyClasses.jar will then be created and can be imported as a Java library.

NOTE: You can read more about JAR at http://java.sun.com/docs/books/tutorial/jar/basics/index.html

---

Resource Files

To specify a resource file, select the Resource Files panel in the Preferences dialog, and then add the resource file.

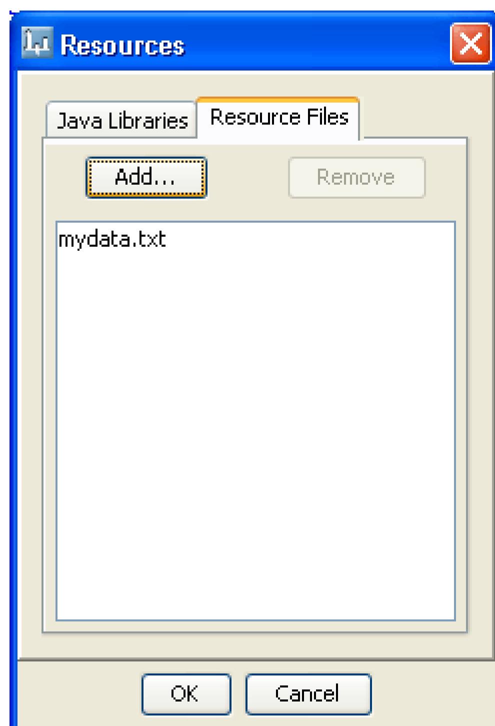

Figure 14: The Resource Files panel in the Preferences dialog

You can now load and access the resource file from the Java object in your Wob class. Create an input stream to the resource file like this:

```
InputStream inputStream = getResource(
<myfilename>).getInputStream();
```

Exercise: Reading text from a resource file

In this example we will add a text file as resource. We will then load the text file and print the text in the console window.

Step 1

Create a text file and add the text "Hello". Save the text file as mydata.txt

Step 2

In the Java object, add the text file as a resource file under “Project > Resources… > Resource Files”

Step 3

Enter the following code into the Java object:

```
package wobs.java.player.extension;

import com.wirefusion.player.*;
import wobs.number.player.NumberWob;
import wobs.bool.player.BoolWob;
import wobs.text.player.TextWob;
import wobs.number2d.player.Number2DWob;
import wobs.color.player.ColorWob;
import java.awt.*;
import java.io.*;

public class JavaWob123 extends wobs.java.player.JavaWob {

  public void init() {
  DataInputStream is = new DataInputStream(
    getResource("mydata.txt").getInputStream());
    try {
      String line = is.readLine();
      while (line != null) {
        System.out.println(line); // Print to Console
        line = is.readLine();
      }
    } catch (IOException e) {e.printStackTrace();}
  }

}
```

NOTE: The class name of the above code will not be accepted by the Java object when you compile the code. When the Java object complains about the class name, choose “Yes”, and the class name will be automatically corrected

---

Step 4

Preview the project by pressing F7 and select the "Show console window" checkbox. "Hello" will be printed in the Console window.

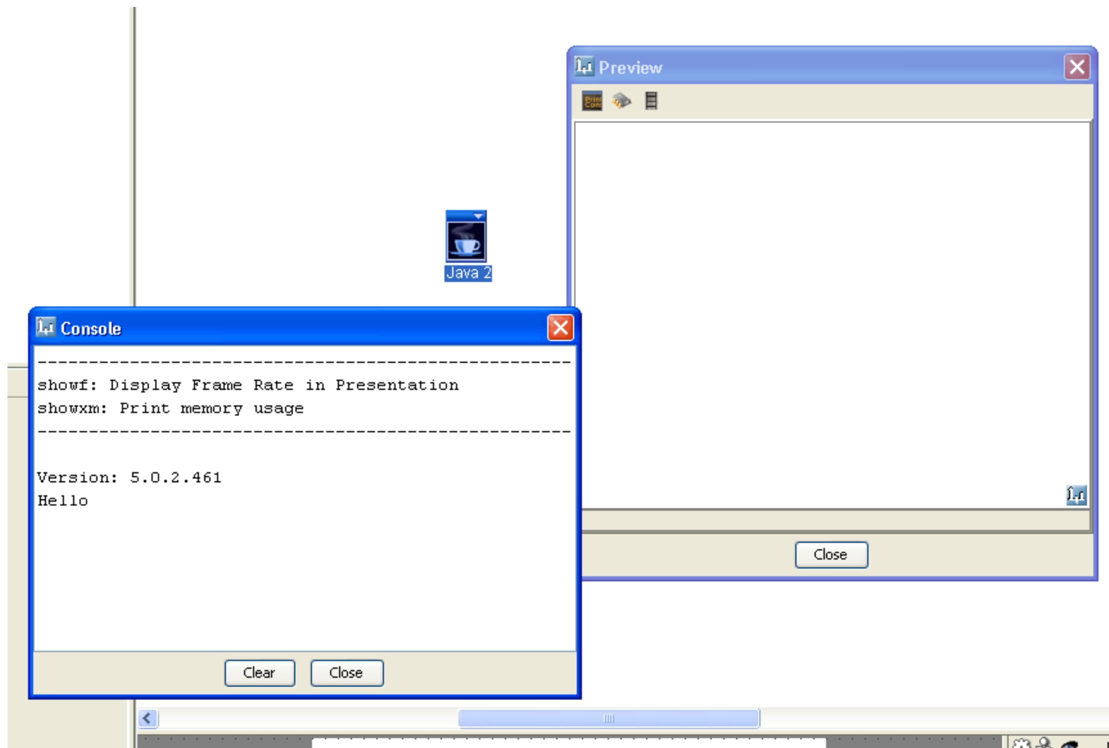

Figure 15: Output in the Console from the example project

If you, after you have added a resource file to a Java object, specify the resource file as dynamic (in the Loading Manager), it will be placed outside of the preload archive (preload.jar). This will allow you to easily edit/replace the file on the web server (assuming you publish the presentation as an applet). If you also add an XML parser as a Java Library, for example NanoXML (see above), you can let the resource file contain XML code and you will be able to parse the XML file from within the Java object.

Notes

- The name of the main class is chosen automatically (JavaWob<class id>) and should not be modified. Each time you open the Java object dialog, its class name will get a new random name.

- To create additional classes to the main class, use private classes defined in the main class file or inner classes.

Part II: WireFusion SDK

Introduction

The WireFusion SDK (Software Development Kit) allows Java programmers to create their own Wobs (WireFusion objects) without some of the limits imposed by the WireFusion Java object. Some of the advantages over the Java object are that you can use your favorite Java development environment, create your own custom configuration GUI and use dynamic project and player resources functions.

Please send comments and feedback regarding this manual or the software to contact@demicron.com

About this chapter

In this chapter we will start by presenting a guide helping you set up your development environment and then try it out with a simple “Hello World” Wob. After that, we will explain in more detail what is needed when creating your own Wob using a “Label” Wob as example.

IMPORTANT: If your development environment uses introspection to show available methods and fields in the API classes, make sure to never use methods not found in the API documentation, since they are subject for change or are for internal use only.

Setting up the environment

Here you will learn how you set up your development environment for Wob development. You will also learn how to create your first "Hello World" Wob using your development environment.

Javac and simple text editor

The following is needed for using *Javac* and a text editor to create Wobs:

- Ant 1.6+ (download from *http://ant.apache.org*)
- Java SDK 1.6+
- Notepad or a similar text editor

Start by downloading *wf5-sdk.zip* from:

http://www.demicron.com/download/docs/v5/wf5-sdk.zip

In the following, it is assumed that you have unpacked *wf5-sdk.zip* to the folder *c:\wirefusion_sdk, but any folder can be used instead*. Below is a step by step test that checks if your WirFusion SDK development environment is functioning:

1. Create the file *c:\wirefusion_sdk\my_wobs\wobs\mycompany_helloworld\player\HelloWorldWob.java* and let it have the following contents:

```
package wobs.mycompany_helloworld.player;

import com.wirefusion.player.Wob;

public class HelloWorldWob extends Wob {
 public void init() {
  System.out.println("Hello World");
 }
}
```

2. Create the file *c:\wirefusion_sdk\my_wobs\my_wobs\wobs\mycompany_helloworld\config.xml* and let it have the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<object name="Hello World" category="MyCompany Objects"
id="mycompany_helloworld" version="1.0"
playerwobclass="wobs.mycompany_helloworld.player.HelloWorldWob">
</object>
```

3. Open a shell window and make *c:\wirefusion_sdk* the current directory. If you for example installed Ant in *c:\apps\apache-ant-1.7.0*, then execute *c:\apps\apache-ant-1.7.0\bin\ant*. You may get a message saying that "tools.jar" could not be located. This message can be ignored. All Wbs in my_wobs and example_wobs will be built and installed by default. What Wobs to build can be specified in build.xml.

4. HelloWorld should now have been installed and available in the Object Library the next time you open WireFusion. Include HelloWorldWob in a new WireFusion project and preview the presentation.
5. If “Hello World” is now printed in the Console window, then your development environment is properly set up!

NOTE: Wobs that are installed into WireFusion directly, like above, will be in Tryout mode when used. This will be fixed in a forthcoming update. Use “release=true” in build.xml and install the object from the File menu to avoid tryout mode.

Eclipse 3.2

Following is a step by step guide showing how the Eclipse environment is set up for Wob development and then a Hello World Wob is developed. It is assumed that you have Eclipse 3.2 and JDK 1.6 or later installed.

NOTE: The JDK must be installed; the JRE only is not enough.

Preparing the development environment

1. Download the WireFusion SDK from:
   http://www.demicron.com/download/docs/v5/wf5-sdk.zip
2. Start Eclipse (using any workspace).
3. In the main toolbar, click the “New Java Project” button (you may have to close the Welcome screen for this button to appear). Enter the project name “wirefusion_sdk”. Select “Create new project in workspace”.
4. Select “Create separate source and output folder” and click “Configure default...”. Select “Folders” and enter “my_wobs” as “Source folder name”. Keep the default value for “Output folder name”. Click “OK”.
5. Close the “New Java Project“ dialog by clicking “Finish”.
6. Select “File > Import... > General > Archive File” and click “Next”. Click “Browse...” and select *wf5-sdk.zip*. The file *build.xml* should be selected in the right column. Enter “wirefusion_sdk “ into the “Into folder:” field. Click “Finish”.
7. The WireFusion class library must be added. Select wirefusion_sdk in the “Package Explorer” view and then select the menu item “Project > Properties”. In the opened dialog, select “Java Build Path” and then select the Libraries tab. Click “Add JARs...”, navigate to the *wirefusion_sdk/lib* folder and select *wfclasses.jar*.
8. An optional step is to add a path to the API doc. Adding this path makes it possible for Eclipse to for example show the correct parameter names in its Auto Completion tools. To do this, download the API documentation from:
   http://www.demicron.com/download/docs/v5/wf5api-doc.zip
   Click the “+” sign to the left of *wfclasses.jar* under the Libraries tab and select “Javadoc in

archive" and enter the location of *wf5api-doc.zip*.

9. Click "OK".

Testing the development environment

To check that your development environment works, we will now create a very basic "Hello World" Wob. We will save the explanations of how general Wobs are made until later.

1. In the main toolbar, click on the "New Java Class" button. Enter HelloWorldWob as class name, `wobs.mycompany_helloworld.player` as package name and `com.wirefusion.player.Wob` as super class, then click "Finish".
2. The source code should now automatically be opened in an editor and look like this:

```
package wobs.mycompany_helloworld.player;

import com.wirefusion.player.Wob;

public class HelloWorldWob extends Wob {

}
```

3. Add the following method to HelloWorldWob:

```
public void init() {
  System.out.println("Hello World");
}
```

4. Press Ctrl+S to save and compile the class.
5. Now you must configure the Wob. This is done in an XML file named *config.xml*. It must be created in the *mycompany_helloworld* package folder (created by Eclipse when the HelloWorldWob class was created). Select the menu item "Window > Open Perspective > Other... > Resource". Right click on the folder *my_wobs/wobs/mycompany_helloworld* in the Navigator view and select the menu item "New > File". Enter the name *config.xml* and click "Finish". The file is now created and automatically opened in the Editor view. Enter the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<object name="Hello World"
category="MyCompany Objects"
id="mycompany_helloworld"
version="1.0"
playerwobclass="wobs.mycompany_helloworld.player.HelloWorldWob">
</object>
```

6. Press Ctrl+S to save *config.xml*
7. Right click on file *build.xml* (not config.xml) in the Navigator view (in the root folder) and select "Run As/Ant Build".
8. HelloWorld should now have been installed and available in the Object Library the next time you open WireFusion. Include HelloWorldWob in a new WireFusion project and preview the presentation.

9. If “Hello World” is now printed in the Console window, then your development environment is properly set up!

Basics of creating a Wob

These are the main parts of creating a Wob:

- Creating the player class (used by the presentation player).
- (optional) Creating the configuration GUI class(es)
- Configuring the Wob in *config.xml*.
- Build and install the Wob (by running *build.xml*).

As an example in the explanations below, a simple Wob will be created that displays a label on the screen. The label text is specified inside WireFusion, but it can also be changed dynamically by sending a Text event to a “Set Label” in-port of the Wob. It is recommended that you have the API Doc of the WireFusion classes at hand while reading the following. The API Doc can be downloaded from:

http://www.demicron.com//download/wf5/wf5api-doc.zip

or viewed directly on the web at:

http://www.demicron.com/wirefusion/v5/docs/api/index.html.

It is recommended that you also study the source code and XML files of the example Wobs, found in the *<WireFusion SDK folder>/example_wobs* folder.

Before you start developing a Wob, you must decide on a Wob ID name. It must be unique for the Wob, otherwise the Wob can’t coexist with other Wobs having the same ID. It is recommended that you use the format *<company name>_<wobname>* (for example *mycompany_helloworld*).

NOTE: The ID must be able to function as a Java package name, so combinations of lower case standard ASCII characters and the '_' character are recommended.

Creating the player class

Now you are ready to create the player class. When you unpacked the SDK, a folder named *my_wobs/wobs* was created. The player source file should be placed in the folder *my_wobs/wobs/<ID>/player folder*, where <ID> is the Wob ID name you decided on above. The class should have the package name “wobs.<ID>.player”. It is recommended that the name of your Wob class ends with the word “Wob”. The Wob class must extend com.wirefusion.player.Wob:

```
package wobs.mycompany_label.player;

import com.wirefusion.player.Wob;

public class MyCompany_LabelWob extends Wob {

}
```

Now we will add a String field that should contain the label text. We will also add the `readData()` method that will read the label value from the player data file (this method is only used if the Wob should have a configuration GUI). A corresponding `writeData()` method will later be created in the configuration GUI class. Some new imports have also been added:

```
package wobs.mycompany_label.player;

import java.awt.*;
import com.wirefusion.player.*;
import java.io.*;

public class MyCompany_LabelWob extends Wob {

  private String label;

  public void readData(DataInputStream dis) {
    try {
      label = dis.readUTF();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Add the following paint method that draws the label on screen:

```
public void paint(Graphics g) {
  g.setColor(Color.black);
  g.setFont(new Font("Dialog", Font.PLAIN, 8));
  g.drawString(label, 10, 5);
}
```

The Graphics class sent to paint is of type `com.wirefusion.player.WfGraphics`. You can cast the Graphics object to `com.wirefusion.player.WfGraphics` to get access to the additional drawing functions.

Creating graphical classes in WireFusion is very similar to creating lightweight Java AWT components. Lightweight AWT components can often easily be converted to Wob classes, sometimes just by extending `com.wirefusion.player.Wob` instead of `java.awt.Component` and by changing the package name.

The final step is to add support for the “Set Label” in-port. The resulting Wob class looks like this:

```
package wobs.mycompany_label.player;

import java.awt.*;
import java.io.*;
import wobs.text.player.*;
import com.wirefusion.player.*;


public class MyCompany_LabelWob extends Wob {

  private String label;

  public void paint(Graphics g) {
    g.setColor(Color.black);
    g.setFont(new Font("Dialog", Font.PLAIN, 8));
    g.drawString(label, 10, 5);
  }

  public void setLabel(String label) {
    this.label = label;
    repaint();
  }

  public void processInportEvent(Wob arg, int inport) {
    super.processInportEvent(arg, inport);
    if (inport == 1) { // Set Label
      String newLabel = ((TextWob) arg).getValue();
      setLabel(newLabel);
    }
  }

  public void readData(DataInputStream dis) {
    try {
      label = dis.readUTF();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Notice the call to `repaint()`, which updates the target area after the label has been modified. In the `processInportEvent()` method above, an inport number (in this case 1) for the “Set Label” inport has been used. Each in-port and out-port of a Wob has an associated unique port number. The port number of the in-port above will be defined later in the Wob configuration file (*config.xml*). You must always call `super.processInportEvent` in the `processInportEvent` method, otherwise the default WireFusion ports will not function.

To create an out-port event, use the `sendX` methods in Wob (see API Doc). For example, to send a Boolean event with value true through port 2, call the `sendBoolean` method like this:

```
sendBoolean("true", 2);
```

Creating the Wob properties class

The GUI of this class is very simple. It will contain a text input field for the label. Start by creating a Wob Properties class that extends `com.wirefusion.project.WobProperties`. Its package must be `wobs.mycompany_label`:

```
package wobs.mycompany_label;

public class MyCompany_LabelProperties
      extends com.wirefusion.project.WobProperties {

}
```

This class can not be compiled, since it does not implement the abstract method `writeData()` of its super class. The `writeData()` method has the responsibility of writing the configuration of the Wob to the player data file (which is read by the player class as we will see later). Let us implement the `writeData()` method. Also, introduce the field “private String label” that should contain the label text:

```
package wobs.mycompany_label;

import java.io.*;

public class MyCompany_LabelProperties
      extends com.wirefusion.project.WobProperties {

  private String label = "Untitled";

  public void writeData(DataOutputStream dos) {
    try {
      dos.writeUTF(label);
    } catch (Exception e) {
     System.out.println(e);
    }
  }
}
```

We want the users to be able to edit the label value. The first step is to create `set` and `get` methods for label. The `set` method MUST support undo, since the general undo function in WireFusion will break if undo is not supported:

```
package wobs.mycompany_label;

import com.wirefusion.undo.*;
```

```
import java.io.*;

public class MyCompany_LabelProperties
     extends com.wirefusion.project.WobProperties {

  private String label = "Untitled";

  public void setLabel(String label) {
    if (!this.label.equals(label)) {
      UndoableStringEdit edit = new UndoableStringEdit(this.label, label) {
        public void setValue(String value) {
          setLabel(value);
        }
      };
      getUndoManager().addEdit(edit);

      String oldLabel = this.label;
      this.label = label;
      firePropertyChange("labelValueChanged", oldLabel, label);
    }
  }

  public String getLabel() {
    return label;
  }

  public void writeData(DataOutputStream dos) {
    try {
      dos.writeUTF(label);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

}
```

The code added to enable the undo support may seem complicated, but it looks almost exactly the same in all types of `set` methods in WobProperties classes. The most notable difference between different `set` methods is that UndoableStringEdit may be of another UndoableEdit class, depending on the argument data type (see API Doc). If the value would have been a Boolean value instead of String value for example, UndoableStringEdit would be replaced by UndoableBooleanEdit and the second argument of `firePropertyChange` would be replaced by `new Boolean(value)`.

What happens in the undo code in `setLabel()` is that a new edit object is instantiated, which holds both the old value and the new value. The object is then added to the undo manager. If the user tells WireFusion to undo, and the last action performed was a “label modification”, the undo manager calls `setLabel(oldValue)` in the Wob, which restores the previous value.

The call to `firePropertyChange` results in that the changed label property is sent to all "property change" listeners. If the input field (see below) of the label property is visible in the GUI, then it will listen to changes of this value (which is identified by the "labelValueChanged" string), and it will update the label value if it is changed (for example if an undo changes the value).

Now it is time to add the code that will result in the label input field. For the Wob properties class to support property editors, it must implement `com.wirefusion.propertyeditor.PropertyEditorCollectionFactory` and the method `createPropertyEditorCollection()` must be implemented (`destroyPropertyEditorCollection()` is already implemented by WobProperties). Also, there are some new imports:

```
package wobs.mycompany_label;

import java.beans.*;
import java.io.*;
import com.wirefusion.propertyeditor.*;
import com.wirefusion.undo.*;

public class MyCompany_LabelProperties
  extends com.wirefusion.project.WobProperties
  implements PropertyEditorCollectionFactory {

...
  public PropertyEditorCollection createPropertyEditorCollection() {
    PropertyEditorCollection propertyEditorCollection = new
      PropertyEditorCollection(this);

    StringPropertyEditor labelEditor = new
      StringPropertyEditor("labelValueChanged",
      "Label",
      "" + getLabel());
    labelEditor.addPropertyChangeListener(new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent evt) {
        String labelValue = (String)evt.getNewValue();
        setLabel(labelValue);
      }
    });
    propertyEditorCollection.addPropertyEditor(labelEditor);

    addPropertyChangeListener(propertyEditorCollection);
    return propertyEditorCollection;
  }
 ...
}
```

In the method `createPropertyEditorCollection()`, the property editors are created. In our example, we only have the label property, so just one property editor, a StringPropertyEditor, is created. Notice that the same identification string as the one in the undo code further above (`labelValueChanged`) is used as argument for the StringPropertyEditor constructor. A property change listener is added to `labelEditor`, and it will change the label value by calling `setLabel()` when the label value is edited.

The `labelEditor` object is added to `propertyEditorCollection`, which is a collection of all the editor objects of this class. The `propertyEditorCollection` object is also added as a listener to property changes (`addPropertyChangeListener(propertyEditorCollection)`). If the label value is changed, a `labelValueChanged` event will be sent (in the `setLabel()` method). The `propertyEditorCollection` will hear the event and send it to `labelEditor` (the correct editor is identified by an ID, in this case `labelValueChanged`).

Again, this may sound complicated, but you can use the code above as a template and reuse almost the same code in all cases. If, for example, the property would have been a Boolean, you would have used BooleanPropertyEditor, otherwise the code would be almost identical.

If more than one property should have property editors, then create one new PropertyEditor object for each property in the `createPropertyEditorCollection()` method and add them to the same `propertyEditorCollection` object.

The last step in the creation of the WobProperties class is to make it *serializable*, so that it can be saved and loaded to/from WireFusion project files. This is done by letting the class extend `java.io.Externalizable` and implementing the methods `readExternal()` and `writeExternal()`. Also, a unique `serialVersionUID` must be added (a random number for example). You can read more about serialization here:

http://java.sun.com/javase/6/docs/technotes/guides/serialization/index.html

After adding the serialization code, the final WobProperties class looks like this:

```
package wobs.mycompany_label;

import java.beans.*;
import java.io.*;
import com.wirefusion.propertyeditor.*;
import com.wirefusion.undo.*;

public class MyCompany_LabelProperties
  extends com.wirefusion.project.WobProperties
  implements PropertyEditorCollectionFactory, Externalizable {

  static final long serialVersionUID = -1526926427652851928L;

  private String label = "Untitled";
```

```
public void setLabel(String label) {
  if (!this.label.equals(label)) {
    UndoableStringEdit edit = new UndoableStringEdit(this.label, label) {
      public void setValue(String value) {
        setLabel(value);
      }
    };
    getUndoManager().addEdit(edit);

    String oldLabel = this.label;
    this.label = label;
    firePropertyChange("labelValueChanged", oldLabel, label);
  }
}

public String getLabel() {
  return label;
}

public PropertyEditorCollection createPropertyEditorCollection() {
  PropertyEditorCollection propertyEditorCollection = new
    PropertyEditorCollection(this);

  StringPropertyEditor labelEditor = new
    StringPropertyEditor("labelValueChanged",
    "Label",
    "" + getLabel());
  labelEditor.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent evt) {
      String labelValue = (String)evt.getNewValue();
      setLabel(labelValue);
    }
  });
  propertyEditorCollection.addPropertyEditor(labelEditor);

  addPropertyChangeListener(propertyEditorCollection);
  return propertyEditorCollection;
}

public void writeData(DataOutputStream dos) {
  try {
    dos.writeUTF(label);
  } catch (Exception e) {
    e.printStackTrace();
  }
}

public void writeExternal(ObjectOutput out) {
```

```
    try {
      super.writeExternal(out);
      out.writeUTF(label);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void readExternal(ObjectInput in) {
    try {
      super.readExternal(in);
      label = in.readUTF();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Configuring the Wob

To configure the Wob, create a file named *config.xml* in the folder *my_wobs/wobs/<ID>*. The contents of a config.xml file have the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<object
  name="<wobname>"
  category="<category>"
  id="<wobid>"
  version="<version>"
  playerwobclass="<player wob classname>"
  wobpropertiesclass="<wob properties classname>"
  wobpropertiespanelclass ="<wob properties panel classname>"
  playerlibs="<playerlibs>"
  projectlibs="<guilibs>"
>
     <targetarea
        modifiespixels="<modifiespixels>"
        viewclassname="<viewclassname>"
        initialwidth="<initialwidth>"
        initialheight="<initialheight>"
     />

     <inports>
        <port id="<portid>"
           name="<portname>"
           docname="<documentation portname>"
           portnumber="<portnumber>"
           portargument="<argument>"
           portinfo="<port description>"
```

```
            autocreate="<auto create>"
         />
         <port .... />
         ...
         <port .... />
      </inports>

      <outports>
         <port id="<portid>"
            name="<portname>"
            docname="<documentation portname>"
            portnumber="<portnumber>"
            portargument="<argument>"
            portinfo="<port description>"
            autocreate="<auto create>"
         />
         <port .... />
         ...
         <port .... />
      </outports>

</object>
```

The object element

The root element must be the object element. It supports the following attributes:

wobname

The Wob name (Example value: “Hello World”).

category

The category you want the Wob to be placed under in the Object Library. If the category doesn't exist, it will automatically be created.

wobid

The unique Wob ID you decided on (Example value: “mycompany_hello_world”).

version

The version of the Wob (Example value: “1.1”).

playerwobclass

The name of the player wob class, including the package name (Example value: “wobs.mycompany_helloworld.player.HelloWorldWob”).

wobpropertiesclass
If the Wob has a WobProperties class, then this value should be its full class name (Example value: "wobs.mycompany_helloworld.MyCompany_LabelProperties").

wobpropertiespanelclass
If a dialog configuration GUI is used as Wob configuration GUI, then this value should be the full name of the WobPropertiesPanel class (Example: wobs.text.TextPropertiesPanel).

playerlibs
A list of class library files (.jar), where each item is separated by a ',' character. The class libraries listed will be included when publishing and available in the player classes. The path should be relative to the wobs/<ID> folder. (Example value: "lib/nanoxml.jar").

projectlibs
A list of class library files (.jar), where each item is separated by a ',' character. The classes in these libraries will be available for "project classes", i.e. the non player classes of the Wob that are used inside WireFusion. The path should be relative to the wobs/<ID> folder. (Example value: "lib/somelib.jar").

The targetarea element

If the Wob should have a Target Area (i.e. is graphical or listens to mouse events), then the object element must contain the targetarea element.

The targetarea element supports the following attributes:

modifiespixels
This value should be true, if the Wob is graphical.

viewclass
If a TargetAreaView class name has been implemented to achieve a custom view of the Target Area inside WireFusion, this value should be the full class name of this class (Example: wobs.image.ImageTargetAreaView)

initialwidth
The initial width of the target area. If left out, a width of 120 will be used.

initialheight
The initial width of the target area. If left out, a width of 90 will be used.

The inports and outports elements

Ports are defined in *config.xml*. In-ports are defined using an <inports> element and out-ports using an <outports> element, which are placed inside the <object> element. Each port is defined with port elements, inside the inports/outports element.

The port elements supports the following attributes:

portid
A unique ID name for the port ("set_label" for example)

name
The port name ("Set Label" for example)

docname
The port name, as it should appear in the auto generated port documentation. If not specified, the name attribute will be used in the documentation. If the value of the *name* attribute uses templates, the *docname* attribute is usefull.

portnumber
The port number, an integer number between 0 and 29000, used when handling incoming port events in the Wob Wob.handleInportEvent class and when creating out-port events.

portargument
The Wob ID of the argument Wob type. For example, if the port uses a Text Wob as argument, then use argument="text". The Wobs commonly used as arguments can be found in *<WireFusion SDK folder>/core_wobs/wobs*. Their Wob ID can be found in their *config.xm*l files. If the port uses no argument, this attribute should be left out.

portinfo
A short description of the port.

autocreate
If true, the port will be automatically created and shown as a Wob port. If this attribute is skipped, true will be used. If false, the port will not be created automatically (can be created manually in the WobProperties class. See WobProperties.createPort() in API Doc).

In our example Wob, *config.xml* looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<object
      name="MyCompany Label"
      category="MyCompany Objects"
```

```
    id="mycompany_label"
    version="1.0"
    wobpropertiesclass="wobs.mycompany_label.MyCompany_LabelProperties"
    playerwobclass="wobs.mycompany_label.player.MyCompany_LabelWob" >

    <targetarea
      modifiespixels="true"
      initialwidth="70"
      initialheight="18"
    />

    <inports>
      <port id="set_label"
        name="Set Label"
        portnumber="1"
        info="Sets the label text"
      />
    </inports>

    <outports>
    </outports>

</object>
```

Help Pages

If you want your Wob to have a help page, then write the help in HTML format in a file named main.html and place it in the following folder (may need to be created): *wobs\<ID>\helpfiles*. It is important that the HTML file starts with exactly the string <HTML><BODY> and ends with </BODY></HTML>.

Example help page:

```
<HTML><BODY>
<p>The Brightness object is a real-time filter, which sets the brightness
level in its Target Area and on all underlying graphics, i.e. graphics in
layers below itself. This object has an alpha channel, which can be used to
control the shape of the brightness effect.</p>
</BODY></HTML>
```

Any images or additional HTML files, to which the main HTML file links, can be placed in the same help files folder.

Building and installing the Wob

As you learned in "Setting up the environment", you run the Ant script *build.xml* to build and install the Wobs in *my_wobs*. The file *build.xml* contains a few configurable parameters that you need to know about:

jdkfolder:
This parameter points at the JDK folder. If the java.home parameter of your system points to a different JDK than you wish to use, then you can change this to point at another JDK.

release:
This is an important parameter. If true, the Wob will be packaged into a .wao file in the "dist" folder of the WireFusion SDK folder. A value of false makes build times faster and the Wob will be directly installed into WireFusion, which makes it suitable to use during development of the Wob. See comment in build.xml for more information.

obfuscaterelease:
If true and if the release parameter is also true, classes will be obfuscated.

wobsfolders:
This property lists your Wob project folders.

Note: When you run build.xml, all Java source files in these folders will be compiled, even if you use the includedwobs and excludedwobs parameters below, these parameters only control which Wobs to process further, into final Wob objects.

includedwobs:
The value "*" means that all wobs in wobsfolders should be built. If you specify a list of Wob ID names, separated by ',', only those wobs will be built.

excludedwobs:
Lists Wob ID names of wobs not to build. Separation character is ','. It is important to always exclude the core Wobs (they are excluded by default), or WireFusion may not run correctly.

Preprocessing:

If you need to perform preprocessing after the classes have been compiled, but before the Wob has been installed or packaged as a .wao file, you can create an ant file named preprocessing.xml in the Wob folder. You can find an example of such a file (obfuscate_using_postprocess.xml), that performs obfuscation, in the resources folder in the WireFusion SDK folder.

File Dependencies

In this section you will learn about handling resources in WireFusion. As an example, we will extend to functionality of the Label Wob constructed above by adding a background image behind the label.

First, the player class is updated with the new feature:

```
package wobs.mycompany_label.player;

import java.awt.*;
import java.io.*;
import wobs.text.player.*;
import com.wirefusion.player.*;

public class MyCompany_LabelWob extends Wob {

  private String label = "Untitled";

  private WfImage image = null;

  public void paint(Graphics graphics) {
    WfGraphics g = (WfGraphics)graphics;
    if (image == null) {
      image = getPlayerResourceById("icon").getWfImage();
    }
    g.drawWfImage(image, 0, 0);
    g.setColor(Color.black);
    g.setFont(new Font("Dialog", Font.PLAIN, 12));
    g.drawString(label, image.getWidth() + 2,
    g.getFontMetrics().getAscent());
  }

  public void readData(DataInputStream dis) {
    try {
      label = dis.readUTF();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void setLabel(String label) {
    this.label = label;
    repaint();
  }

  public void processInportEvent(Wob arg, int inport) {
    super.processInportEvent(arg, inport);
    if (inport == 1) { // Set Label
```

```
            String newLabel = ((TextWob) arg).getValue();
            setLabel(newLabel);
        }
    }
}
```

In the `paint` method, the image is loaded if it has not been loaded before, and it is painted in the top left corner of the Wob.

The WobProperties class must be modified so that it contains a file property, and it must create a file dependency for the current image file, so that the image file will be included in presentations. The default image file to be used is the file *<WireFusion SDK folder>/images/reddot.gif* (a 32x32 GIF image with a red dot). The resource files of a Wob is always placed in the *wobs/<ID>* folder. Copy *reddot.gif* to the folder *<WireFusion SDK folder>my_wobs\wobs\mycompany_label*.

After the modification of MyCompany_LabelProperties, it will look like below:

```
package wobs.mycompany_label;

import java.beans.*;
import java.io.*;
import com.wirefusion.propertyeditor.*;
import com.wirefusion.undo.*;
import com.wirefusion.project.*;

public class MyCompany_LabelProperties extends
        com.wirefusion.project.WobProperties implements
        PropertyEditorCollectionFactory, Externalizable {

    static final long serialVersionUID = -1526926427652851928L;

    private String label = "Untitled";

    public void init() {
        if (getPlayerResourceById("icon") == null) {
            File defaultImageFile = getFileInWobFolder("reddot.gif");
            File file = addPlayerResource(defaultImageFile, "icon", true,
                    LOAD_BEFORE_START);
        }
    }

    public void setLabel(String label) {
        if (!this.label.equals(label)) {
            UndoableStringEdit edit = new UndoableStringEdit(
                    this.label, label) {
                public void setValue(String value) {
                    setLabel(value);
```

```
            }
        };
        getUndoManager().addEdit(edit);

        String oldLabel = this.label;
        this.label = label;
        firePropertyChange("labelValueChanged", oldLabel, label);
    }
}

public String getLabel() {
    return label;
}

public void setImageFile(File file) {
    if (!getPlayerResourceById("icon").getName().
        equals(file.getName())) {
        UndoableFileEdit edit =
            new UndoableFileEdit(getImageFile(), file) {
            public void setValue(File aFile) {
                MyCompany_LabelProperties.this.setImageFile(aFile);
            }
        };
        getUndoManager().addEdit(edit);
        File oldFile = file;
        addPlayerResource(
            file, "icon", true, WobProperties.LOAD_BEFORE_START);
        firePropertyChange("fileChanged", oldFile, file);
    }
}

public File getImageFile() {
    return getPlayerResourceById("icon");
}

public PropertyEditorCollection createPropertyEditorCollection() {
    PropertyEditorCollection propertyEditorCollection =
        new PropertyEditorCollection(this);

    StringPropertyEditor labelEditor = new StringPropertyEditor(
            "labelValueChanged", "Label", "" + getLabel());
    labelEditor.addPropertyChangeListener(new PropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent evt) {
            String labelValue = (String) evt.getNewValue();
            setLabel(labelValue);
        }
    });
    propertyEditorCollection.addPropertyEditor(labelEditor);
```

```
            File defaultFolder = new File(
               WobUtilities.USER_RESOURCES_FOLDER, "images");
            FilePropertyEditor fileEditor = new FilePropertyEditor(
               "fileChanged",
               "Image",
               getImageFile(),
               new String[] { "gif", "jpg", "jpeg", "png" },
               null,
               defaultFolder,
               "image",
               true);
            fileEditor.addPropertyChangeListener(new PropertyChangeListener() {
               public void propertyChange(PropertyChangeEvent evt) {
                  File file = (File) evt.getNewValue();
                  setImageFile(file);
               }
            });
            propertyEditorCollection.addPropertyEditor(fileEditor);

            addPropertyChangeListener(propertyEditorCollection);
            return propertyEditorCollection;
         }

         public void writeData(DataOutputStream dos) {
            try {
               dos.writeUTF(label);
            } catch (Exception e) {
               e.printStackTrace();
            }
         }

         public void writeExternal(ObjectOutput out) {
            try {
               super.writeExternal(out);
               out.writeUTF(label);
            } catch (Exception e) {
               e.printStackTrace();
            }
         }

         public void readExternal(ObjectInput in) {
            try {
               super.readExternal(in);
               label = in.readUTF();
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
      }
```

In the `init` method, the default image file is added as a player resource, if the Wob is just created. A `setImageFile()` and a `getImageFile()` method is added. A file editor is added in the `createPropertyEditorCollection()` method. Also, the methods `writeData()`, `readExternal()` and `writeExternal()` are updated so that they read and write the image filename.

Custom configuration dialogs

A custom configuration dialog can be created by extending `com.wirefusion.project.WobPropertiesPanel`. The GUI is added to the WobPropertiesPanel class and will be displayed when the user click on the “Configure...” button in the Properties view (the General tab). Also, the `wobpropertiespanelclass` attribute in *config.xml* must specify the full WobPropertiesPanel class name. To learn more, see API Documentation and the Button example Wob, available in *<WireFusion SDK folder>/example_wobs*.

Custom Target Area View

Custom contents can be displayed in the Target Area of a Wob. To do this, extend the class TargetAreaView. The attribute `viewclass` of the `targetarea` element in config.xml must have the full TargetAreaView classname as value. To learn more, see API Doc and the Button example Wob, available in *<WireFusion SDK folder>/example_wobs*.

Add-on Builder

When you run *build.xml* with the directinstall property in *build.xml* set to false, each Wob built will be stored as a WireFusion Add-on Object file (.wao). These Wobs can be added to WireFusion by choosing File > Install Add-on…

There is also a second type of Add-on, with the following features:

- Support for serial numbers.
- Custom splash images are shown during the installation process and when the installation is finished.
- A single Add-on can contain multiple Wobs.
- Custom resource files can be added to the WireFusion Application Data folder.
- A *purchase URL* can be shown in the serial number dialog.
- Tryout versions of Add-ons requiring a serial number can be installed.

The suffix for Add-on files of this type is .wad (WireFusion Add-on).

We will explain how to create an Add-on of the second, more advanced type, through an example Add-on, located in the folder *<WireFusion SDK folder>/addon_projects/example_addon_project*. The Add-on includes the Invert filter (available as an example Wob in *<WireFusion SDK folder>/example_wobs*) and it requires a serial number for non tryout installation.

The Add-ons are configured in an *addon.properties* file, containing the following parameters:

name:
The name of the Add-on.

version:
The Add-on version.

includedwobs:
Comma separated list of Wobs that the Add-on should contain. The Wobs are specified with the Wob ID and must be available for compilation by the main build.xml Wob builder script.

serialnumbervalidator:
The serial validator class name (see below). If omitted, serial number check will not be used.

purchaseurl:
Links to web page, from where to obtain serial number (if serialnumbervalidator is specified). This is an optional property.

In the example, *addon.properties* has the following configuration:

```
name=Example Add-on
version=1.0
includedwobs=invert
serialnumbervalidator=serialnumbervalidator.ExampleSerialNumberValidator
purchaseurl=http://www.mysite.com/store
```

Serial Number validation

If your Wob is commercial, you may want users to have to enter a serial number to install the Wob (as non Tryout). To enable serial number validation, create a subclass of `com.wirefusion.sdk.SerialNumberValidator` (see API doc) and implement the method `validateSerialNumber()`. The class must be placed in the `serialnumbervalidator` package, which should be placed in the Add-on project folder. This is the source code for the serial number validator class in our example:

```
package serialnumbervalidator;
```

```
import com.wirefusion.sdk.*;

public class ExampleSerialNumberValidator extends SerialNumberValidator {
  pubic boolean validateSerialNumber(String serialnumber) {
    if (serialnumber.equals("123"))
      return true;
    else
      return false;
  }
}
```

The only serial number that this Add-on will accept is “123”.

Splash Screens

Two JPEG images must be placed in the Add-ons folder, *mainsplash.jpg* and *finishedsplash.jpg*. They must have the size 350x450 pixels. The first image, *mainsplash.jpg*, will be shown during the installation of the Add-on and *finishedsplash.jpg* when the installation is finished.

License text

A text file named *License.txt*, containing the license text of the Add-on, should be placed in the Add-on folder (you may use the example License text freely, but you may want to modify “MyCompany” and “MyCountry”).

Building

To build the Add-on, the Ant script *build_addon.xml*, located in the folder *<WireFusion SDK folder>/addon_projects/example_addon_project*, is executed. The resulting Add-on is generated in the *dist* folder inside the Add-on folder.

When you create your own Add-on projects, the Add-on project folders must be placed in the *addon_projects* folder. When you have created all the required resource files and configured *addon.properties*, then just copy *build_addon.xml* from this example into your Add-on folder and execute it.

NOTE: You may have to modify the jdkfolder property of *build_addon.xml*.

Part III: WireFusion 3D API

Introduction

The WireFusion 3D API (Application Program Interface) allows advanced users to reach and control parameters dynamically in a 3D scene while running a WireFusion 3D presentation.

The programming and connection to the 3D scene is done with the built-in WireFusion Java object. Selected nodes and fields can be dynamically changed, allowing the control of material properties, objects, cameras, lights, textures, positions, rotations, animations etc. It is also possible to dynamically add and remove 3D models and textures to running presentations.

Please send comments and feedback regarding this manual or the software to contact@demicron.com

About this chapter

In this chapter we will introduce the WireFusion 3D API, explain the basics of how to control a 3D model and scene, and also go through some basic exercises.

Requirements

User Requirements

To take full advantage of the 3D API it is recommended that you know the Java programming language, or any other programming language. Non-programmers can also take advantage of the 3D API by re-using already made code or by copying and pasting code into the WireFusion Java object.

If you've never used WireFusion before, then it is highly recommended that you work through the tutorial ***Getting Started, Volume I***, which requires no former WireFusion knowledge. You should also read the WireFusion ***Java*** manual, which explains how to write Java code in WireFusion using the Java object.

The full 3D API can be found on the Demicron web site (www.demicron.com/wirefusion/api)

System Requirements

In order to use the 3D API you have to have either the WireFusion Enterprise version 5.0 (or higher) or the WireFusion Educational edition version 5.0 (or higher).

Definitions

WireFusion W3F Format

In order to control your 3D scene dynamically using the 3D API you need to know some basic things about the X3D and VRML format, and about the W3F format (WireFusion 3D Format), which is the 3D format used internally in WireFusion. When an X3D or a VRML file is imported to WireFusion, then it is automatically converted to the WireFusion 3D Format (.w3f).

X3D and VRML Definitions

Some X3D and VRML definitions.

Scene Graph

The scene graph contains a hierarchy of ***nodes***, which describe objects and their properties.

Nodes

Nodes are the fundamental component of a ***scene graph*** and are used to represent real-world objects. Each node in a scene graph is an instance of existing ***node types***. Nodes contain ***fields***.

Node Types

A characteristic of each ***node*** that describes, in general, its particular ***field*** semantics. Each ***node type*** has a fixed set of ***fields***. For example, Shape, Material, DirectionalLight and TouchSensor are node types (see supported node types below).

Fields

A property or attribute of a ***node***. ***Fields*** may contain various kinds of data and one or many values. Each field has a default value.

```
DEF Box01 Transform {
  translation -1.397 0.6373 0
  rotation -1 0 0 -1.571
  children [
    Shape {
      appearance DEF _1_-_Default Appearance {
        material Material {
          diffuseColor 0.5882 0.5882 0.5882
          ambientIntensity 1.0
          specularColor 0 0 0
          shininess 0.145
          transparency 0
        }
        texture ImageTexture {
          url "CARPTBLU.JPG"
        }
      }
      geometry DEF Box01_FACES(1) IndexedFaceSet {
        coord DEF Box01_COORD Coordinate { point [
          -0.9935 0 0.9841, 0.9935 0 0.9841, -0.9935 0 -0.9841,
          -0.9935 1.068 -0.9841, 0.9935 1.068 -0.9841 ] }
        coordIndex [
          0, 2, 3, -1, 3, 1, 0, -1, 4, 5, 7, -1, 7, 6, 4, -1, 0, 1, 5, -1
          2, 0, 4, -1, 4, 6, 2, -1 ]
      }
    }
  ]
}
DEF Sphere01 Transform {
  translation 1.472 -0.02812 0
  rotation -1 0 0 -1.571
  children [
    Shape {
```

Figure 16: Part of a VRML code, specifying the scene graph

WireFusion 3D API Definitions

The representation of the ***scene graph*** in the WireFusion 3D API is as follows.

Scene Graph

The X3D/VRML ***scene graph*** is represented in the 3D API with a hierarchy of `X3DNode` classes. The `X3DScene` class extends `X3DNode` and acts as the root node for the scene graph.

Nodes

Every X3D/VRML ***node*** is represented in the 3D API as an instance of a `X3DNode` class.

Node Types

All X3D/VRML ***node types*** are represented in the 3D API with the same `X3DNode` class. The `X3DNode` class can be queried for the ***node type*** it contains.

Fields

Every X3D/VRML ***field*** is represented in the 3D API as an instance of a `X3DField` class.

NOTE: The full 3D API can be found at the Demicron site (www.demicron.com/wirefusion/3dapi.html).

Basics

You access the ***scene graph*** through a `Browser` object, which can be retrieved by using the getParent- and getChild methods provided in the class `Wob`. In the below examples we refer to the VRML code seen in Figure 17.

```
DEF Box01 Transform {
  translation -1.397 0.6373 0
  rotation -1 0 0 -1.571
  children [
    Shape {
      appearance DEF _1_-_Default Appearance {
        material Material {
          diffuseColor 0.5882 0.5882 0.5882
          ambientIntensity 1.0
          specularColor 0 0 0
          shininess 0.145
          transparency 0
        }
        texture ImageTexture {
          url "CARPTBLU.JPG"
        }
      }
      geometry DEF Box01_FACES(1) IndexedFaceSet {
        coord DEF Box01_COORD Coordinate { point [
          -0.9935 0 0.9841, 0.9935 0 0.9841, -0.9935 0 -0.9841,
          -0.9935 1.068 -0.9841, 0.9935 1.068 -0.9841 ] }
        coordIndex [
          0, 2, 3, -1, 3, 1, 0, -1, 4, 5, 7, -1, 7, 6, 4, -1, 0, 1, 5, -1
          2, 0, 4, -1, 4, 6, 2, -1 ]
      }
    }
  ]
}
DEF Sphere01 Transform {
  translation 1.472 -0.02812 0
  rotation -1 0 0 -1.571
  children [
    Shape {
      appearance USE _1_-_Default
```


Figure 17: VRML code

To retrieve a 3D scene `Browser` for a particular 3D Scene object that is dragged into the Script Area, you access the 3D Scene Wob and cast it to a `Browser`. Depending on where the Java object with the java code is located relative to the 3D Scene object, you combine `getParent` and `getChild` method calls to access the 3D Scene Wob. If you for example have the Java object located in the same folder as a 3D Scene object named “Box 3D Scene”, you would call:

```
Browser boxBrowser = (Browser) getParent().getChild("Box 3D Scene");
```

Once you have the `Browser` object, you can call getScene() to get the `X3DScene` class containing the ***scene graph.***

```
X3DScene boxScene = boxBrowser.getScene();
```

Since X3DScene extends X3DNode, you can use the methods found in X3DNode to access the nodes of the scene graph. To access any named node in the scene graph you use the getNode(String) method as follows:

```
X3DNode boxTransformNode = boxScene.getNode("Box01");
```

You can now access the fields of a node by calling the getField(int fieldIndex) method. This method takes an integer as parameter, which indicates what field to retrieve. In the 3D API there is a package named **wobs.scene3d.nodes** containing classes, one for each node type, where constants for all their fields are found. For instance, to access the field ***translation*** in a Transform node we would do:

```
X3DField translationField =
     boxTransformNode.getField(Transform.translation);
```

In the X3DField class there is a collection of get and set methods to be used to access the contained java type or object. Every X3DField object has a type that specifies what X3D/VRML field type it represents, and this decides what functions you can call. For instance, by checking the Transform node in the 3D API we see that translation is of type *SFVec3f*. After checking the conversion table in class X3DField we know this corresponds to a float array with three elements, so to retrieve the Java instance of the translation field we can use the getFloatArray() method as follows:

```
float[] translationFloatArray = translationField.getFloatArray();
```

We can now for example read the z-position by accessing its second element:

```
float z = translationFloatArray[2];
```

To set the field to a new value we can use the set(Object newValue) method, and as a parameter use a float array with three elements. We could for instance change the translationFloatArray and call set() with the translationFloatArray as parameter like this:

```
translationField.set(translationFloatArray);
```

NOTE: You need to be very careful when using the set and get methods for a field. Calling the wrong function on the field will result in an error.

Returning to the boxTransformNode, we can access the nodes in its children field by calling:

```
X3DField childrenField =
     boxTransformNode.getField(Transform.translation);
X3DNode[] childrenNodes = childrenField.getNodeArray();
```

The same thing can however be achieved more compactly by calling the getChildren() method of a X3DNode:

```
X3DNode[] childrenNodes = boxTransformNode.getChildren();
```

Checking the X3D/VRML-file we know that the first child node is a Shape node:

```
X3DNode shapeNode = childrenNodes[0];
```

We can now access the Appearance node found in the appearance field by:

```
X3DNode appearanceNode =
      shapeNode.getField(Shape.appearance).getNode();
```

Or we could use the shorter `getNode(int fieldIndex)` method instead:

```
X3DNode appearanceNode = shapeNode.getNode(Shape.appearance);
```

If we wanted to change the transparency of this Shape to 50 percent, we would continue to retrieve the material node and the transparency field:

```
X3DNode materialNode =
      appearanceNode.getNode(Apperance.material);
X3DField transparancyField =
      materialNode.getField(Material.transparency);
```

Transparency is of type *SFFloat* which means we can use the `set(float floatValue)` method to set the field.

```
transparencyField.set(0.5f);
```

Get the browser

To get the Browser for a 3D Scene object found in the same folder as your Java object you call:

```
Browser browser = (Browser) getParent().getChild("name of 3D Scene Object");
```

Get the scene

To get the scene for a 3D Scene object you must first access the Browser and then call:

```
X3DScene scene = browser.getScene();
```

Get a named node

To get a named node in the scene graph you call:

```
X3DNode namedNode = yourX3DNode.getNode("node name");
```

Get a field

To access a field of a node you call the `getField(int fieldIndex)` method in `X3DNode`. This method takes as parameter an integer, which indicates what field to retrieve. In the 3D

API there is a package named "nodes" containing classes, one for each node type, where constants for all their fields are found.

```
X3DField yourField = yourX3DNode.getField(int fieldIndex);
```

Read and change a field

To read or change a field you use one of the get or set methods found in `X3DField`. Every `X3DField` object has a type that shows what X3D/VRML field type it represents, and this type specifies what functions you can call. To learn what X3D/VRML field type a field represents, check the nodes package in the 3D API. In the 3D API description of `X3DField` there is a conversion table between X3D/VRML types and WireFusion Java types, which is used to see what method calls are applicable. For example, the Boolean java type represents fields of type SFBool and you use these two methods to read and change their value:

```
boolean booleanValue = yourX3DField.getBoolean();
yourX3DField.set(booleanValue);
```

Listening to field changes

You can register to receive events when a field value has been changed. This mechanism can be useful when there is a need to process fields of certain nodes, like the TouchSensor's over and active fields.

To receive field events you add a `X3DfieldEventListener` to the field. By implementing the readableFieldChanged-method for the interface you will receive a `X3DFieldEvent` every time the field has changed. Below example shows how you would do to print out the over status of a TouchSensor:

```
X3DField field = touchSensorNode.getField(TouchSensor.over);
field.addFieldEventListener(new X3DFieldEventListener() {
  public void readableFieldChanged(X3DFieldEvent e) {
    System.out.println("Over status: "+e.getSourceField().getBoolean());
  }
});
```

Load new textures

To load a texture you must first access the Browser and then call:

```
Object newTexture = browser.loadTexture("optionalImageFileName",
"optionalAlphaFileName");
```

To replace the texture in a Texture node you would then call:

```
textureNode.getField(Texture.image).set(newTexture);
```

Load an external 3D file

To load a new external 3D file must first access the Browser and then call:

```
X3DScene newScene = browser.loadScene("filename.w3f");
```

Add and remove a 3D file

To add a new X3D/VRML file you first load it through the Browser by calling:

```
X3DScene newScene = browser.loadScene("filename.w3f");
```

To add it to another scene you do:

```
scene.addChild(newScene);
```

You can remove it by doing:

```
scene.removeChild(newScene);
```

NOTE: You have to convert your X3D/VRML files to the WireFusion 3D Format (file extension *.w3f*) before loading. This is done from the 3D Scene object's dialog (Save or Convert buttons).

Replace a 3D file

To replace an X3D/VRML file you must first access the Browser and then call:

```
browser.replaceScene(newX3DScene);
```

Basic Excercises

Exercise: Adding a 3D file to a 3D scene

In this exercise, we will dynamically load and add a 3D teapot to a running 3D scene, which contains a 3D table. This will be achieved using the 3D API and we will place the teapot on the table. We will also learn how to remove the teapot again from the 3D scene.

Step 1 – Load the teapot

Insert a 3D Scene object into a new and empty project.

From the **3D Type** dialog, choose to load a **3D Object**.

Load the file *teapot.wrl*, found in the directory *[Path]/My Documents/WireFusion/resources/3D models/teapot/*

NOTE: Remember that any possible changes of the default rendering settings you make for the teapot will not be seen in the final presentation, as it is the master 3D scene's default rendering settings that will be visible, i.e. the table scene, and not the dynamically loaded teapot scene.

Step 2 – Save the teapot in WireFusion 3D format

In the 3D Scene dialog, press the **Save** button and store your model on *C:* on your hard disk (Figure 18).

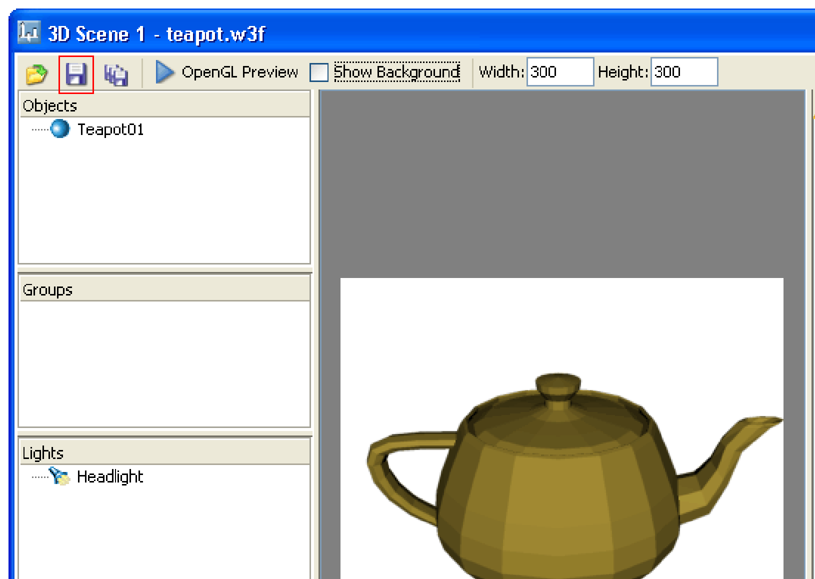

Figure 18: Saving the teapot as .w3f

NOTE: When loading and adding a 3D model dynamically to another 3D scene, using the 3D API, then you first have to manually convert the 3D model (i.e. the model you want to add) to the WireFusion 3D Format (*.w3f*). This is done by loading the 3D model into the 3D Scene dialog and then pressing the save button.

Step 3 – Load the table

Now, in the 3D Scene dialog, press the Replace button (Figure 19) and load the file *table.wrl*, found in the directory *[Path]/My Documents/WireFusion/resources/3D models/table/*

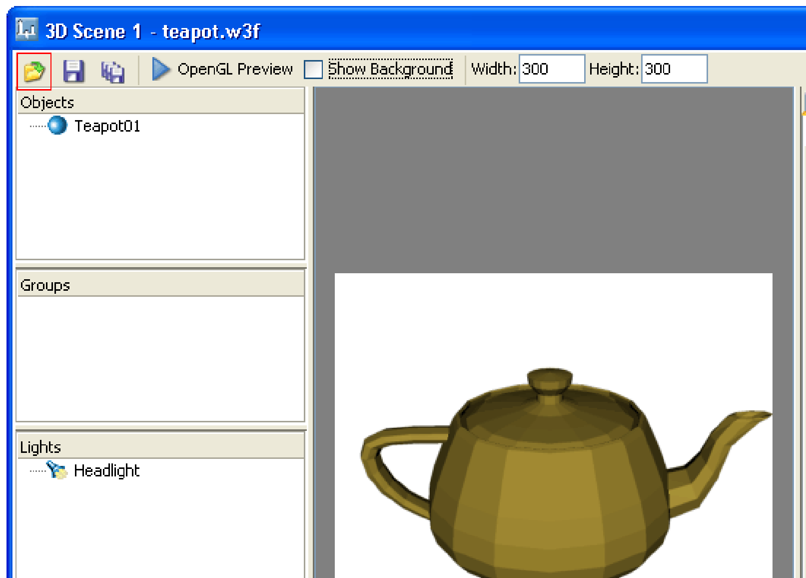

Figure 19: Replacing the teapot model

Make sure to unmark all the checkboxes in the Replace options popup dialog (Figure 20).

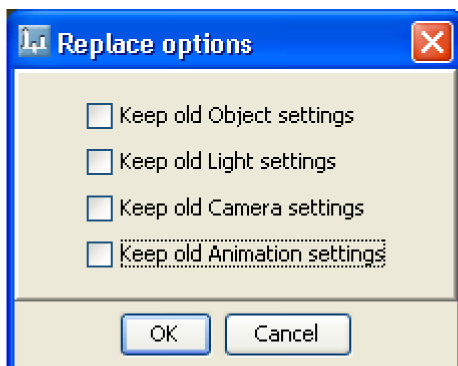

Figure 20: Unmarking all checkboxes in the Replace option dialog

NOTE: You could also have deleted the teapot 3D Scene object and inserted a new one, and then loaded the table object. The replacement procedure is not necessary for this project, it is just faster.

Step 4 – Close the table dialog

When the table model is loaded into the 3D Scene, without doing any changes, press the OK button to close the 3D Scene dialog.

Resize the 3D Scene object's Target Area so that it fits the stage dimension.

Press Alt + S on your keyboard to do this automatically.

NOTE: We will use the table scene as the master scene and the default rendering settings you make for the table will also be effective for the 3D scenes you will load/add dynamically.

Step 5 – Insert a Java object and create in-ports

Insert a Java object to your project.

IMPORTANT: When working with the 3D API make sure to import the wobs.scene3d.player package, the wobs.scene3d.nodes package and the wobs.scene3d.nodes.Shape class. See Figure 21.

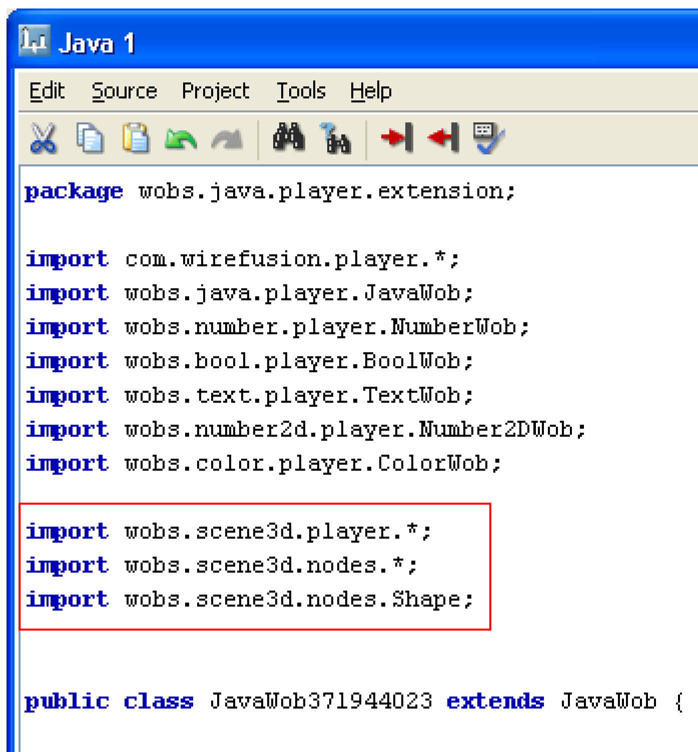

Figure 21: Importing the 3D node classes

In the Java code, insert an in-port. To do this, choose the menu option Tools > Insert In-port code (Figure 22).

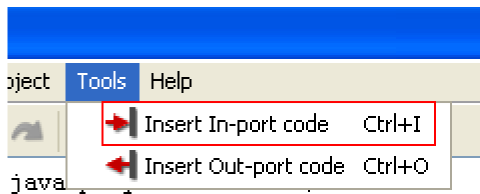

Figure 22: Starting the in-port code generator

When the In-port code generator dialog opens, choose Any Argument as argument, and then choose the port name addTeapot (Figure 23).

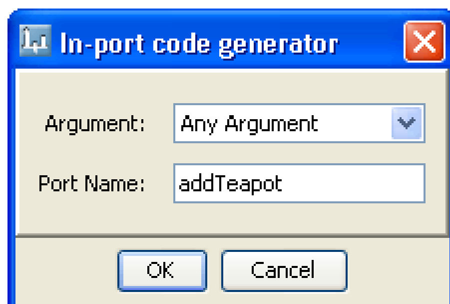

Figure 23: In-port code generator window

Repeat the above and create another in-port, but name this port to removeTeapot instead.

```
import wobs.scene3d.nodes.Shape;

public class JavaWob917757203 extends JavaWob {
  public void inport_addTeapot() {

  }
    public void inport_removeTeapot() {

  }
  public void init() {
  }
```

Figure 24: The in-ports added to the code

Step 6 – Adding a scene

Still in the Java object, create a field, named `teapotScene` of type `X3DScene` as below:

```
private X3DScene teapotScene;
```

Add the below code into the `inport_addTeapot` method:

```
if (teapotScene!=null) return;
Browser browser = (Browser) getParent().getChild("3D Scene 1");
X3DScene tableScene = browser.getScene();
teapotScene = browser.loadScene("file:///c:/teapot.w3f");
teapotScene.move(0, 85, 0);
teapotScene.scale(0.4f, 0.4f, 0.4f);
tableScene.addChild(teapotScene);
```

- The first line is used to see if we already have added the teapot to the table scene.
- The second line is used to access the browser for the 3D Scene object containing the table scene. Since we didn't rename the 3D Scene object the name is the automatically generated name "3D Scene 1".
- The third line is used to access the table scene.
- The fourth line will load the saved teapot scene (Step 2) from your hard drive. We saved it on C:\ to be able to test the functionality from inside WireFusion. You need to replace this line to the correct path before you publish the presentation. If you place the file teapot.w3f in the published project directory, then you should replace the line with: `teapotScene = browser.loadScene("teapot.w3f");`
- The fifth and sixth lines are used to correctly position and scale the teapot scene so it fits into the table scene.
- The seventh and final line will add the teapot scene as a child to the table scene.

Step 7 – Removing a scene

Add the code provided below into the `inport_removeTeapot` method:

```
if (teapotScene==null) return;
Browser browser = (Browser) getParent().getChild("3D Scene 1");
X3DScene tableScene = browser.getScene();
tableScene.removeChild(teapotScene);
teapotScene=null;
```

- The first line is used to see if the teapot is added to the table scene.
- The second line is used to access the browser for the 3D Scene object containing the table scene.
- The third line is used to access the table scene.
- The fourth line will remove the teapot scene from the table scene.
- The fifth line clears the teapot variable, which indicates that it is not added anymore.

Press the OK button to close the Java object dialog.

Step 8 – Insert Buttons

Insert a Button object. When its dialog opens, choose the button label Add (Figure 25).

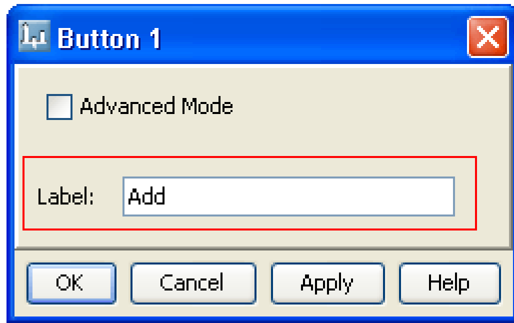

Figure 25: Changing the button label to Add

Insert another Button object. When its dialog opens, choose the button label Remove.

Step 9 – Connect the Button objects to the Java object

Connect:

Button 1 > Out-ports > Button Clicked

to

Java 1 > In-ports > addTeapot

Connect:

Button 2 > Out-ports > Button Clicked

to

Java 1 > In-ports > removeTeapot

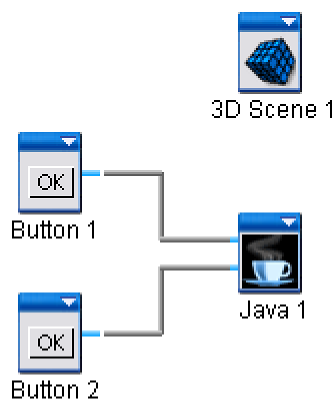

Figure 26: The two Button objects connected to the Java object

Step 10 – Preview

OK, done! If you have followed the steps above, saved the file *teapot.w3f* on *C:* and used the link *file:///c:/teapot.w3f* in your Java code, then it should work to add (and remove) the teapot to (and from) the table.

Press F7 to preview (Figure 27).

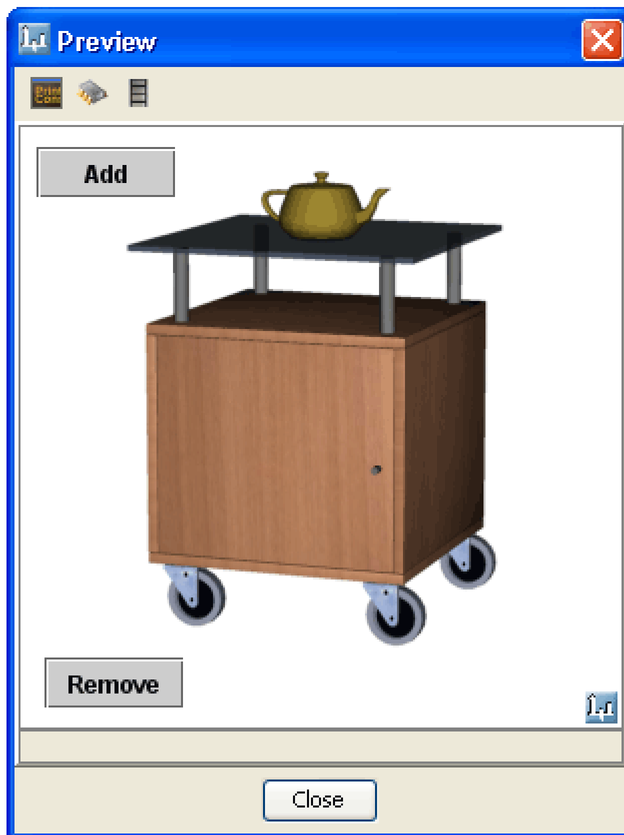

Figure 27: Preview with the teapot added

NOTE: If you publish the presentation, remember to change the link in the code (in step 6, line 4) to teapot.w3f, and copy the file to the folder containing your published project.

Exercise: Changing the texture of an object

We will continue with the above example and will now add the possibility to dynamically load and replace a texture, using the 3D API.

Step 1 - Create in-ports

In the Java object, insert a new in-port, Tools > Insert In-port code

When the In-port code generator dialog opens, choose Any Argument as argument, and then choose the port name changeTexture.

Step 2 – Replacing the texture

Create a field in your code named `texture` of type `X3DScene` as below:

```
private Object texture;
```

Add the below code into the `inport_changeTexture` method:

```
Browser browser = (Browser) getParent().getChild("3D Scene 1");
X3DScene tableScene = browser.getScene();
X3DNode doorTransformNode = tableScene.getNode("Door");
X3DNode[] doorChildren = doorTransformNode.getChildren();
X3DNode shapeNode = null;
  for (int i=0; i<doorChildren.length; i++) {
    if (doorChildren[i].type==Node.Shape) {
      shapeNode = doorChildren[i];
      break;
    }
  }
X3DNode appearanceNode = shapeNode.getNode(Shape.appearance);
X3DNode textureNode =
appearanceNode.getNode(Appearance.texture).getNodeArray(MultiTexture.texture)[0];
texture = browser.loadTexture("file:///c:/ashsen.jpg","");
textureNode.getField(Texture.image).set(texture);
```

- The first line is used to access the browser for the 3D Scene object containing the table scene.
- The second line is used to access the running table scene.
- The third line retrieves the node named Door.
- The fourth line retrieves the children nodes of the door Transform node.
- The lines 5-11 scans through the children nodes to retrieve the Shape node under the door transform.
- The 12th line retrieves the Appearance node under the Shape node.
- The lines 13-14 retrieve the Texture node under the Appearance node. It is assumed that the texture node is the first element of the X3DNode array in the MultiTexture.texture field.
- The 15th line loads the texture from the local hard drive. This line should, before publishing the presentation, be replaced with the line: `texture = browser.loadTexture("ashsen.jpg","");`
- The 16th line retrieves and sets the field image of the Texture node.

Press the OK button to close the Java dialog.

Step 3 – Insert a Button

Insert a Button object. When its dialog opens, choose the button label Texture.

Step 4 – Connect the Button objects to the Java object

Connect:

Button 3 > Out-ports > Button Clicked

to

Java 1 > In-ports > changeTexture

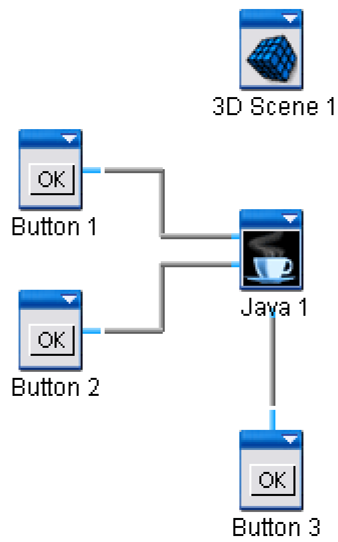

Figure 28: The replace texture button connected to the Java object

Step 5 – Preview

OK, you are done! If you have followed the steps above, have located the file *ashen.jpg* on *C:* and used the link *file:///c:/ashen.jpg* in your Java code, then it should work to replace the texture on the table door.

Press F7 to preview (Figure 29).

Figure 29: Preview with the door texture changed

NOTE: If you publish the presentation, remember to change the link in the code (in step 2, 14th line) to *ashen.jpg*, and copy the file to the folder containing your published project.

Appendix: Supported Nodes and Fields

Out of the 36 X3D/VRML nodes supported in WireFusion, twelve of them are useful when doing advanced 3D programming. Each of these twelve nodes and their supported fields are explained below.

For a full explanation of the X3D and VRML nodes and fields, please refer the X3D and VRML97 specification at the Web3D Consortium web site (www.web3d.org).

Appearance

The Appearance node specifies the visual properties of geometry.

Supported fields in the Appearance node are:

material

The *material* field contains a *Material* node.

texture

The *texture* field contains a *Texture* node.

textureTransform

The *textureTransform* field contains a *textureTransform* node.

DirectionalLight

The DirectionalLight node defines a directional light source that illuminates along rays parallel to a given 3-dimensional vector.

The *direction* field specifies the direction vector of the illumination emanating from the light source in the local coordinate system. Light is emitted along parallel rays from an infinite distance away. A directional light source illuminates only the objects in its enclosing parent group. The light may illuminate everything within this coordinate system, including all children and descendants of its parent group. The accumulated transformations of the parent nodes affect the light.

Supported fields in the DirectionalLight node are:

ambientIntensity

Specifies the ambient intensity, in the range of 0 to 1. The ambient intensity specifies the intensity of the ambient emission from the light.

color

Specifies the light color. Colors are specified with hexadecimal values (0xRRGGBB).

direction

Specifies the light direction.

intensity

Specifies the light intensity, in the range of 0 to 1.

on

Turns the light on and off.

Group

A Group node contains children nodes without introducing a new transformation. It is equivalent to a Transform node containing an identity transform.

Supported fields in the Group node are:

children

Stores the children nodes of this object. Used to access children nodes.

About Grouping Nodes

Grouping nodes have a field that contains a list of children nodes. Each grouping node defines a coordinate space for its children. This coordinate space is relative to the coordinate space of the node of which the group node is a child. Such a node is called a *parent* node. This means that transformations accumulate down the scene graph hierarchy.

Supported grouping nodes:
- Group
- Transform

Supported children nodes:
- DirectionalLight
- Group
- PointLight
- Shape
- TimeSensor
- TouchSensor
- Transform
- Viewpoint

Material

The Material node specifies surface material properties for associated geometry nodes and determines how light reflects off an object to create color.

Supported fields in the Material node are:

ambientIntensity

Specifies the ambient intensity, in the range of 0 to 1. The *ambientIntensity* field specifies how much ambient light from light sources this surface shall reflect. Ambient light is omni directional and depends only on the number of light sources, not their positions with respect to the surface.

diffuseColor

Specifies the material diffuse color. Colors are specified with hexadecimal values (0xRRGGBB). The *diffuseColor* field reflects all light sources depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects.

emissiveColor

Specifies the material emissive color. Colors are specified with hexadecimal values (0xRRGGBB). The *emissiveColor* field models "glowing" objects.

reflectionMap

The *reflectionMap* field contains a *Texture* node that is used for the reflection. This is field is a WireFusion specific and is not found in X3D/VRML.

shininess

Specifies the material shininess, in the range of 0 to 1. Shininess specifies a material specular scattering exponent.

specularColor

Specifies the material specular color. Colors are specified with hexadecimal values (0xRRGGBB). The *specularColor* and *shininess* fields determine the specular highlights (e.g., the shiny spots on an apple). When the angle from the light to the surface is close to the angle from the surface to the viewer, the *specularColor* is added to the diffuse and ambient colour calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.

textureOpacity

Specifies the texture opacity, in the range of 0 to 100. The *opacity* field specifies how "clear" a textur is, with 0 being completely transparent, and 100 completely opaque. This is field is a WireFusion specific and is not found in X3D/VRML.

transparency

Specifies the material transparency, in the range of 0 to 1. The *transparency* field specifies how "clear" an object is, with 1 being completely transparent, and 0 completely opaque.

PointLight

The PointLight node specifies a point light source at a 3D location in the local coordinate system. A point light source emits light equally in all directions; that is, it is omnidirectional. PointLight nodes are specified in the local coordinate system and are affected by ancestor transformations.

Supported fields in the PointLight node are:

ambientIntensity

Specifies the ambient intensity, in the range of 0 to 1. The ambient intensity specifies the intensity of the ambient emission from the light.

color

Specifies the light color. Colors are specified with hexadecimal values (0xRRGGBB).

intensity

Specifies the light intensity, in the range of 0 to 1.

location

Specifies the light location.

on

Turns the light on and off.

radius

Specifies the light radius, in the range of 0 to infinity. A PointLight node illuminates geometry within the *radius* of its *location.*

Shape

The Shape node has two fields, *appearance* and *geometry* (not supported), which are used to create rendered objects in the world. The *appearance* field contains an Appearance node that specifies the visual attributes (e.g., material and texture) to be applied to the geometry. The *geometry* field contains a geometry node. The specified geometry node is rendered with the specified appearance nodes applied.

Supported fields in the Shape node are:

appearance

The *appearance* field contains an *Appearance* node.

Texture

The Texture is a merger of the X3D/VRML ImageTexture and PixelTexture nodes.

Supported fields in the Texture node are:

image

Specifies the texture. Takes a WireFusion Texture.

TextureTransform

The TextureTransform node defines a 2D transformation that is applied to texture coordinates. This node affects the way textures coordinates are applied to the geometric surface.

Supported fields in the TextureTransform node are:

center

Specifies the center position for the texture coordinates. The *center* field specifies a translation offset in texture coordinate space about which the *rotation* and *scale* fields are applied.

rotation

Specifies the rotation angle for the texture. The *rotation* field specifies a rotation in radians of the texture coordinates about the *center* point after the scale has been applied. A positive rotation value makes the texture coordinates rotate counterclockwise about the center, thereby rotating the appearance of the texture itself clockwise.

scale

Sets the scale factor for the texture coordinates. The *scale* field specifies a scaling factor of the texture width and height about the *center* point. The *scale* value shall be in the range of negative infinity to positive infinity.

translation

Sets the translation vector of the texture coordinates.

TimeSensor

The TimeSensor node is used to drive animations.

Supported fields in the TimeSensor node are:

fraction

Specifies the animation fraction in the range of 0 to 1. Original X3D/VRML node name is *fraction_changed*.

TouchSensor

A TouchSensor node tracks the location and state of the pointing device and detects when the user points at geometry contained by the TouchSensor node's parent group.

Supported fields in the TouchSensor node are:

enabled

Turns the TouchSensor node on and off. If the TouchSensor node is disabled, it does not track user input or send events.

hitPoint

Contains the 3D point on the surface of the underlying geometry, given in the TouchSensor node's coordinate system. Original X3D/VRML node name is *hitPoint_changed*.

hitTextCoor

Contains the texture coordinates. Original X3D/VRML node name is *hitTextCoord_changed*.

active

Signals TRUE when the primary button is pressed over the TouchSensor and FALSE when it is released. When the field signals TRUE, it grabs all further motion events from the mouse until it is released and sets the *active* field to FALSE (other TouchSensors' will not generate events during this time). Original X3D/VRML node name is *isActive*.

over

Signals True if the mouse cursor is within (or in contact with) the TouchSensor node's geometry. When True it cause the TouchSensor node to generate *active* events (e.g., by pressing the primary mouse button). Original X3D/VRML node name is *isOver*.

Transform

The Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors.

Supported fields in the Transform node are:

center

Specifies a translation offset from the origin of the local coordinate system (0,0,0).

children

Stores the children nodes of this object. Used to access children nodes.

rotation

Specifies a rotation of the coordinate system. The *rotation* field specifies a rotation in radians.

scale

Specifies a non-uniform scale of the coordinate system. *scale* values shall be greater than zero.

scaleOrientation

Specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The *scaleOrientation* applies only to the scale operation.

translation

Specifies a translation to the coordinate system.

About Grouping Nodes

Grouping nodes have a field that contains a list of children nodes. Each grouping node defines a coordinate space for its children. This coordinate space is relative to the coordinate space of the node of which the group node is a child. Such a node is called a *parent* node. This means that transformations accumulate down the scene graph hierarchy.

Supported grouping nodes:
- Group
- Transform

Supported children nodes:
- DirectionalLight
- Group
- PointLight
- Shape
- TimeSensor
- TouchSensor
- Transform
- Viewpoint

Viewpoint

The Viewpoint node defines a specific location in the local coordinate system from which the user may view the scene.

Supported fields in the Viewpoint node are:

bound

Specifies the viewpoint when True is received. Original X3D/VRML node name is *set_bind*.

fieldOfView

Specifies the field of view (FOV) value, in the range of 0 to infinity.

orientation

Sets the rotation relative to the default orientation. In the default position and orientation, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up.

position

Sets the position relative to the coordinate system's origin (0,0,0). In the default position and orientation, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up.

Index

3

3D API 52
- basics 57
- definitions 54
- description 52
- supported Nodes and Fields 72

A

Add-on Builder 49
- building 51
- license text 51
- serial number validation 50
- splash screens 51

Appearance Fields
- material 72
- texture 72
- textureTransform 72

Appearance Node 72

Argument
- Insert In-port code 12
- Insert Out-port code 13

Auto Generate Ports 12

AWT
- components 15
- events 14

D

Definitions
- W3F 54, 55
- VRML 54
- X3D 54

Development environment, SDK
- preparing 29
- testing 30

DirectionalLight Fields
- ambientIntensity 72
- color 73
- direction 73
- intensity 73

DirectionalLight Node 72

E

Eclipse 29

Examples
- 3D API
  - add and remove a 3D file 61
  - get a field 59
  - get a named node 59
  - get the browser 59
  - get the scene 59
  - listening to field changes 60
  - load an external 3D file 60
  - load new textures 60
  - read and change a field 60
  - replace a 3D file 61
- Java object
  - Adding a Pop-up menu 15
  - Create an out-port 11
  - Creating an In-port 9
  - sendColor 11
- SDK
  - building and installing the Wob 44
  - file dependencies 45
  - help page 43

Exercises
- 3D API
  - adding a 3D file to a 3D scene 62
  - changing the texture of an object 68
- Java object
  - Adding a Text field 19
  - Reading text from a resource file 24

F

Fields 54, 56
- Appearance 72
- DirectionalLight 72
- Group 73, 74
- PointLight 75
- Shape 76
- Texture 76
- TextureTransform 76
- TimeSensor 77
- TouchSensor 77
- Transform 78
- Viewpoint 79

G

Group Fields
- children 73

Group Node 73

Grouping Nodes 73

I

In-ports 9

Insert In-port code 8, 12

Insert Out-port code 8, 12

J

Java Libraries 22

Java object
- development environment 7
- requirements 5
- resources 22
- shortcuts 6

Java Object 2

Javac 28

M

Material Fields
- ambientIntensity........74
- diffuseColor........74
- emissiveColor........74
- reflectionMap........74
- shininess........74
- specularColor........74
- textureOpacity........75
- transparency........75

Material Node........74

N

Node Types........54, 55
Nodes........54, 55
- Appearance........72
- DirectionalLight........72
- Group........73
- Material........74
- PointLight........75
- Shape........76
- Texture........76
- TextureTransform........76
- TimeSensor........77
- TouchSensor........77
- Transform........78
- Viewpoint........79

O

Out-ports........10

P

Player events........14
PointLight Fields
- ambientIntensity........75
- color........75
- intensity........75
- location........75
- on........75
- radius........75

PointLight Node........75
Port Name
- Insert In-port code........12
- Insert Out-port code........13

Preferences, Java object........8

R

Requirements
- 3D API........53
- Java object........5

Resource Files........23
Resources, Java object........22

S

Scene Graph........54, 55
SDK........27
- Add-on Builder........49
- building and installing the Wob........44
- configuration dialogs........49
- configuring the Wob........39
- creating a Wob........31
- development environment........29
- environment........28
- file dependencies........45
- help pages........43
- inports elements........42
- license text........51
- object element........40
- outports elements........42
- player class........31
- serial number validation........50
- splash screens........51
- Target Area view........49
- targetarea element........41
- Wob properties class........34

Serial number validation........50
Shape Fields
- appearance........76

Shape Node........76
Shortcuts
- Java object........6

Splash screens........51
Supported Nodes and Fields........72
System Events........14
- AWT events........14
- Player events........14

System requirements
- 3D API........53
- Java object........5

T

Texture Fields
- image........76

Texture Node........76
TextureTransform Fields
- center........76
- rotation........76
- scale........76
- translation........77

TextureTransform Node........76
TimeSensor Fields
- fraction........77

TimeSensor Node........77
TouchSensor Fields
- active........77
- enabled........77
- hitPoint........77
- hitTextCoor........77
- over........77

TouchSensor Node........77
Transform Fields
- center........78
- children........78
- rotation........78
- scale........78
- scaleOrientation........78
- translation........78

Transform Node........78

U

User requirements
  3D API 53
  Java object..5

V,W

W3F
  Fields..56
  Node Types 55
  Nodes 55
  Scene Graph 55
Verify Source 8
Viewpoint Fields
  bound..79
  fieldOfView...79
  orientation 79
  position 79
Viewpoint Node... 79
WireFusion 3D API..................................... 52
WireFusion 3D Format................................ 54
WireFusion SDK 27
VRML
  Fields 54
  Node Types.. 54
  Nodes... 54
  Scene Graph... 54

X

X3D
  Fields 54
  Node Types.. 54
  Nodes... 54
  Scene Graph... 54

* * *

Publication date: November 7, 2007

The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

All rights reserved © 2007 Demicron AB, Solna, Sweden. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Demicron and its licensors, if any.

Trademarks — Demicron and WireFusion are trademarks of Demicron AB. Acrobat, Flash, Photoshop are registered trademarks of Adobe Systems Incorporated. Java, SunSoft are trademarks of Sun Microsystems, Inc. Windows95, Windows98, Windows ME, Windows NT, Windows 2000, Windows XP, Windows Vista are trademarks or registered trademarks of Microsoft Corporation. Pentium is a trademark of Intel Corporation. OS X is a trademark of Apple Computer. All other trademarks are the property of their respective owners.