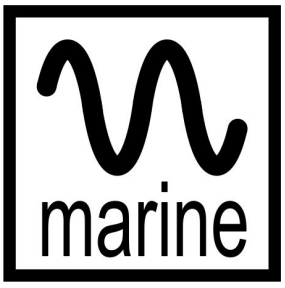

a framework for multimedia artistic interactivity experimentation

Developers Guide

Version 1.1.1 | January/2018 | Ricardo Scholz

This release is susceptible to architectural changes which may not allow backward compatibility. This project is part of my PhD Research at Universidade Federal de Pernambuco, Brazil. For further information, please, write to "contact at marineframework dot org".

Ricardo Scholz

Contents

I. Legal Statement

marine is licensed under a Creative Commons Attribution Non-Commercial Share Alike license, and it uses several third-party softwares and libraries. Each of them must be used in compliance with their terms of use or license conditions.

Please, when using or extending *marine*, make sure you read its Terms of Use (check for the most up to date version at www.marineframework.org), as well as its third-party softwares Terms of Use (copy available in Anex I).

1. Environment Preconditions

1.1. Installing Kinect SDK 2.0

Download Kinect SDK 2.0 and follow install instructions:
https://www.microsoft.com/en-us/download/details.aspx?id=44561

1.2. Installing EyesWeb for Windows

According to the official website, *"EyesWeb is an open software research platform for the design and development of real-time multimodal systems and interfaces. (...) EyesWeb is conceived, designed and developed by InfoMus Lab. (...) EyesWeb is copyright (c) Laboratorio di Infomatica Musicale - DIST - University of Genoa."*

marine portable version includes EyeWeb files packaged. However, for development purposes, it may be interesting to install EyesWeb in your machine. Follow the steps below to install it:

1. Download EyesWeb for Windows, version 5.5.0 or later, from Casa Paganini/InfoMus webpage: http://www.infomus.org/eyesweb_eng.php

2. Follow install instructions.

3. After downloading *marine* source code (section 2.2), make sure to properly update EyesWeb install path and executable in the configuration file of *marine* source code (`config\system.config`); it is important to notice that values in this file follow Java String rules, so backslash path separator must be escaped by another backslash and values containing blank spaces must be enclosed by double quotes; for instance, a valid path would be:

```
eyesweb.install.directory="C:\\Program Files (x86)\\EyesWeb 5.5.0"
```

If you use an absolute path for EyesWeb install directory, remember to set the line below, in the same file:

```
eyesweb.external=true
```

4. After install is complete, run the following executable, in order to test whether your installation is working:

```
"<<EyesWeb installation directory>>\EywConsole.exe"
```

1.3. J4K Library

The J4K library is part of the University of Florida Digital Worlds (ufdw.jar) Java library. It was developed by Prof. Angelos Barmpoutis, and extended by students and faculty of the SAGE program, at the University of Florida Digital Worlds Institute. UFDW library comes embedded on *marine*. There is **no need** to download or install it.

1.4. Processing

Processing is *“a flexible software sketchbook and a language for learning how to code within the context of the visual arts”*. It is free to download and open source. You do **not need** to install Processing 3 in your machine to run *marine*. Processing *jar* comes embedded with the source code.

2. Development Environment Setup

2.1. Minimum Java SDK Version

The minimum JDK version required is Java SDK 8. Install instructions can be found at:
http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html

2.2. Maven Configurations

The following instructions assume you have Maven properly installed and configured. For further information about installing Maven, see https://maven.apache.org.

If you are using **Eclipse IDE**, and want to directly download the project as a Maven Project, make sure you have the M2E (Maven Integration for Eclipse) installed. On Eclipse, go to "Help >> Eclipse Marketplace" and search for it.

2.2.1. Downloading Core Project

marine core project is hosted by BitBucket, at the following address:
https://bitbucket.org/ricardoscholz/marine

On Eclipse IDE (with M2E plugin installed), follow the steps bellow:

1. Go to "File >> New >> Project…";
2. Look for "Maven >> Checkout Maven Projects from SCM", and click "Next";
3. Fill the field "SCM URL" with the values "git" and "https://<<youruser>>@bitbucket.org/ricardoscholz/marine.git"
4. Click "Next";
5. Choose the folder on which the source code will be download or just let it be downloaded on your default workspace, and click "Finish".

2.2.2. Adjusting Core Project Details

In order to run the Core Project, some details must be adjusted, after downloading source code. In Eclipse IDE:

1. Go to "Project >> Properties >> Java Compiler" and make sure the project uses a Java version equal or greater to 1.8, update 60;
    a. If your JRE version is lower than this, go to "Project >> Properties >> Java Build Path", delete the JRE System Library and add a proper JRE System Library (from the "Add Library…" button).

The second step is only needed if you want to run *marine* from within your project, for test purposes:

2. Go to "Project >> Properties >> Java Build Path" and add the following libraries, from the "lib" folder of the project, by pressing "Add Jar":
    a. gluegen-rt-natives-windows-amd64.jar
    b. j4k-natives-windows-amd64.jar

c. j4k2-natives-windows-amd64.jar
d. jogl-all-natives-windows-amd64.jar
e. jogl-all.jar *(under investigation, as it should be added from the pom.xml)*

2.2.3. Installing Core Library on Local Maven Repository

On Eclipse IDE, execute the following commands, with a right click over the project name:

1. Run as… >> Maven Clean
2. Run as… >> Maven Build
    a. When executing a Maven Build for the first time, a popup window will show up; just fill the “Goals” field with “package”, press “Apply” and “Run”.

After that, you might end up with a JAR file on the target folder of your project.

Then, right click over the project name and select “Run as… >> Maven Install”, or open command prompt and enter the following command:

```
mvn install:install-file
-Dfile=”{core-project-path}\target\{jar-filename}.jar”
-DgroupId=mustic.scholz.marine -DartifactId=marine-core
-Dversion={core-jar-filename-version} -Dpackaging=jar
```

For instance:

```
mvn install:install-file
-Dfile=”c:\git\marine-core\target\marine-core-1.1.1.jar”
-DgroupId=mustic.scholz.marine -DartifactId=marine-core -Dversion=1.1.1
-Dpackaging=jar
```

3. Running an Example

3.1. Starting a Streaming of Data

3.1.1. From an Actual Kinect Device

Just connect your Kinect device on the USB 3.0 port.

3.1.2. From a Pre-recorded File

1. Open Kinect SDK 2.0;
2. Open a pre-recorded file;
3. Go to the “Play” tab;
4. Click “Connect to Service” button: 
5. Click ”Play” button to start streaming: 
6. If you want the streaming to run several times, set the “Loop: Count” to a value greater than one.

3.2. Starting an Example Application in the Core Project

1. Open file `mustic.scholz.marine.test.SkeletonTest.java`;

2. Run it as a Java Application; performance may take some seconds to start playing; you should see the performers’ skeleton on the projection screen;

3. Use “esc” to exit the performance (if focus is on projection screen), or terminate the java thread directly from the IDE;

4. After stopping the application, open Task Manager and kill “EywConsole.exe (32 bit)” process (as many as you find, just in case you have forgotten to kill it on previous executions).

a. Manually killing “EywConsole.exe (32 bit)” is not the normal behaviour, and is needed only if you interrupt the main *marine* thread before it gets the chance to kill this process; using `PerformanceFacade.destroy()` will kill the process properly.
b. If your program is intended to run directly from the core module, without an interface, it is interesting to map a key to call `PerformanceFacade.destroy()`.

4. Marine Architecture

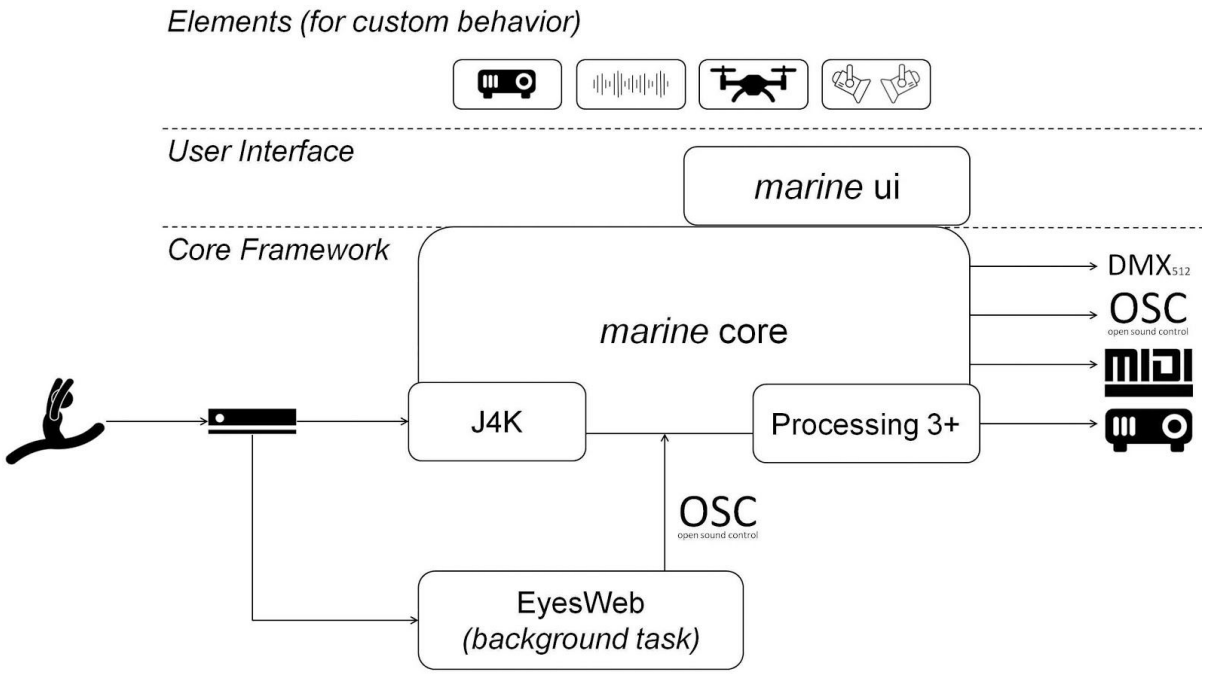

Figure 1. *High level architecture organization.*

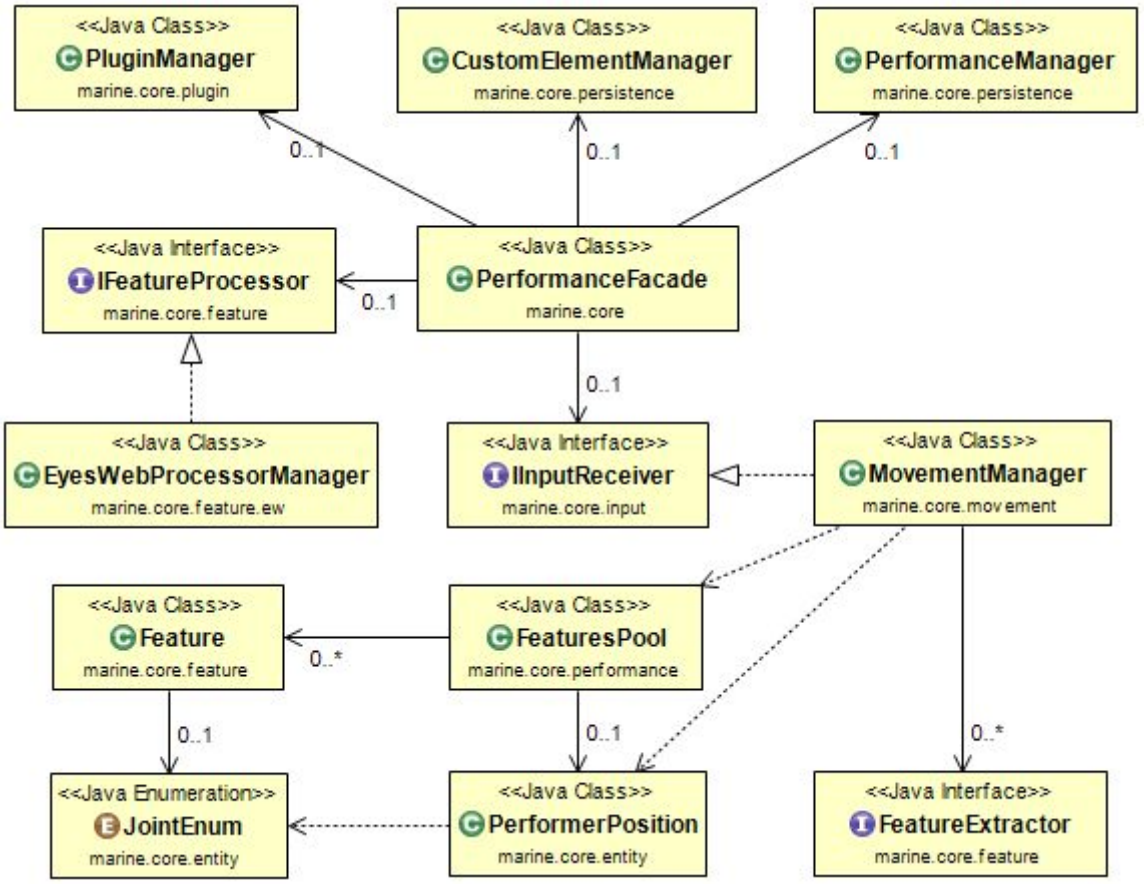

Figure 2. *marine core internal modules, as of version 1.0.0 (“mustic.scholz” omitted on package names)*

4.1. Performance Facade

The `PerformanceFacade` is a Singleton which provides all operations of the framework, which either can be called through an UI or through a main class. It contains references for the internal modules and delegates functions. Functions are divided into 4 groups:

1. Playback functions, such as “play”, “pause”, “stop”, “move forward”, “move backward”, delegated to the `Performance` instance;
2. Elements’ table management functions, such as “add element”, “remove element”, “add transition”, “remove transition”, also delegated to the `Performance` instance;
3. Performance management functions, such as “new performance”, “open performance” and “save performance”, delegated to the `PerformanceManager` instance;
4. Library management functions, such as “save as custom element”, “load custom element”, “load element”, delegated to the `PluginManager` or `CustomElementManager` instances;

As it is a Singleton, the facade is accessible from its `getInstance()` method. To start playing a performance, the `play()` method must be called. Before you leave the application, make sure to call `destroy()` method, as it performs actions to free resources being used, and calls the `IFeatureProcesssor.stop()` method, which kills the EyesWeb process (in case you are using the default implementation, `EyesWebProcessorManager`).

You can determine which screen will be used for projection, as well as the default background color and the projection plane through the method:

```
public void prepareProjection(int backgroundColor, int projectionScreen, ProjectionPlane plane);
```

As projection screen information need to be set through `PApplet.settings()` method, the projection screen can only be changed before the first run of a performance (when PApplet is actually started).

4.2. Performance Module

The `Performance` class is a `PApplet` (see Processing 3+ documentation at https://processing.org/), responsible for painting stuff on a second screen. It runs asynchronously to the facade thread, updating and painting the elements registered on its elements’ table.

The `Performance` class contains an `ElementTable`, which manages the elements at each point of execution. Element tables may contain a number of layers, each layer may run a custom element at a time. All layers run in parallel (not a real parallelism, as they run in a same thread). A transition changes the active elements of each layer to the next element. It

is possible to navigate through the elements forward and backward. Only one column of the table is played at a time, however all layers are played simultaneously.

4.3. Performance Manager

Performance manager runs synchronously to the facade thread and provides methods to access and save previously built performances as files in the file system. Persistence related classes and interfaces are in the package “scholz.dance.core.persistence”. A binary serialization implementation has been implemented, and is available under the package “mustic.scholz.marine.core.persistence.binary”.

To save the current performance, the following method must be called:

```
PerformanceFacade.savePerformance(String fileUrl);
```

To open a previously saved performance as the current performance managed by the facade, call:

```
PerformanceFacade.openPerformance(String fileUrl);
```

The known drawbacks of the current approach are:

1. Selection parameters must provide a parameter serializer in their constructor, to allow serialization of its generic type;
2. There is no serialization protocol versioning; therefore, elements which evolve and need to change their serialization protocol may not be backward compatible.

4.4. Plug-in Manager

Plug-in Manager runs synchronously to the facade thread and provides methods to manage plug-ins. Plug-in Elements are elements developed by third party developers and packed as plug-ins (jar files renamed to *.mar) which can be loaded by the end user. Once a plug-in is first imported, *marine* will uncompress the Plug-in JAR file in the folder specified by the property “`system.plugins.directory`” in the system.config file. If the folder already exists, the plug-in manager will skip the importation and show a console message (there is no overwriting, to avoid messing up with previous performances).

To import a plug-in JAR file, call:

```
PerformanceFacade.importElementPlugin(String fileUrl) throws Exception;
```

To load all element plug-ins previously imported, call:

```
List<ElementPlugin> PerformanceFacade.loadAllElementPlugins();
```

To remove a previously imported plug-in, use:

```
PerformanceFacade.delete(ElementPlugin plugin);
```

Remember that deleting a plug-in will make it unavailable for new performances, and previously saved performances will not be able to be loaded until the plug-in is imported again (under the same filename). Future versions may allow saving a performance embedding the plugins used.

4.5. Custom Elements Manager

Library manager runs synchronously to the facade thread and provides methods to manage custom elements. Custom elements are any customizations performed on a given element parameters which can be saved for future use in a customized elements' bank.

Custom elements are saved under a pair of keys: the element name and the color used to identify that specific configuration.

To load all custom elements, or a specific custom element, previously saved, use one of the following:

```
List<CustomElement> PerformanceFacade.loadAllCustomElements();

CustomElement loadCustomElement(String directory, Integer key);
```

There are several ways to save a customized element:

```
void PerformanceFacade.saveAsCustomElement
      (Element element, String name, int key, String directory);

void PerformanceFacade.saveCustomElements
      (List<CustomElement> customElements);

void saveCustomElement(CustomElement customElement);
```

Finally, deleting a custom element may be achieved by calling:

```
void delete(CustomElement element);
```

4.6. Feature Processor

The `IFeatureProcessor` interface defines a contract for the module which will be responsible for silently starting the thread or external program which will compute features to be sent to the input listeners registered with the `IInputReceiver`. Though, it is expected that this module runs asynchronously to the facade.

The default implementation is the `EyesWebProcessorManager`, which starts EyesWeb and executes the features computation patch. EyesWeb is executed as an independent process, called through command line. In order to kill EyesWeb process, it is necessary to call `IFeatureProcessor.stop()` method, what can be done by calling the method

`PerformanceFacade.destroy()`. The default implementation runs asynchronously to the facade.

If you exit the application without calling any of these methods, you may have to kill EyesWeb process manually, on windows task manager. If many of these instances are running at the same time, a lot of processing power will be wasted, increasing the response time of the entire machine.

4.7. Input Receiver

The `IInputReceiver` interface defines a contract for the module responsible for starting the module which will process the input received from the Feature Processor (features and performer position messages). It is expected that this module runs asyncrhonously to the facade.

The default implementation is the `MovementManager`, which process input received through OSC messages by the EyesWeb process and performer positions received directly from the MS Kinect, using J4K SDK, updating the `FeaturesPool` (features and performer positions). It also manages the `FeatureExtractor` instances registered, so that customized features are always updated when a new performer position is received.

5. Writing Custom Element Plug-ins

5.1. The Template Marine Element Project

5.1.1. Downloading the Template Project

In order to develop your own elements, you can start a new project from scratch, or you can make a copy of the Template Marine Element Project, which is an empty project, already setup for a new element development.

The Template Marine Element Project is available in the Downloads page at Marine site, as well as Marine Agents Project at Bitbucket, under the folder named “template”. Follow the steps on Section 2.2.1 and Section 2.2.2, substituting the target address by the following address:

https://bitbucket.org/ricardoscholz/marine-agent/template

5.1.2. Setting Up Project Details

Once you have the Template Marine Element Project zip file and extracted its contents to a folder of your preference, or you have downloaded it directly through Git from BitBucket, create a new project in Eclipse IDE and:

1. Go to “Project >> Properties >> Java Compiler” and make sure the project uses a Java version equal or greater to 1.8, update 60;
    a. If your JRE version is lower than this, go to “Project >> Properties >> Java Build Path”, delete the JRE System Library and add a proper JRE System Library (from the “Add Library…” button).

2. Right click on project, then “Maven >> Update Project”, uncheck all checkboxes, except for:
    a. Update dependencies
    b. Force Update of Snapshots/Releases
    c. Refresh workspace resources from local filesystem
    d. Clean projects
3. Go to “Run >> Run as… >> Maven Clean”;
4. Go to “Run >> Run as… >> Maven Build”, fill the Goals field with “package” (without the quotes).

    IMPORTANT: if you want to run performances or tests within your custom element project, you must go to “Project >> Properties >> Java Build Path” and add the following libraries, from the *marine core project* “lib” folder, by pressing “Add Jar”:
    a. gluegen-rt-natives-windows-amd64.jar
    b. j4k-natives-windows-amd64.jar
    c. j4k2-natives-windows-amd64.jar
    d. jogl-all-natives-windows-amd64.jar
    e. jogl-all.jar

5.2. Elements Lifecycle

Elements are classes which run inside a `PApplet` (see Processing 3+ documentation at https://processing.org/). Their lifecycle is very similar to a `PApplet`. A `setup()` method is executed once, at the beginning of element execution, in order to load all resources needed. Then, separate `update()` and `paint()` methods (instead of a `draw()` method, used in the `PApplet` approach) are executed every cycle, until an external interruption is performed. When it occurs, the `destroy()` method is called to free resources. The maximum frame rate is set to 30 fps.

A method `fireCommand(int command, Object… params)` will be called asynchronously when focus is in the projection screen (`PApplet` screen) and keys 0 to 9 are pressed.

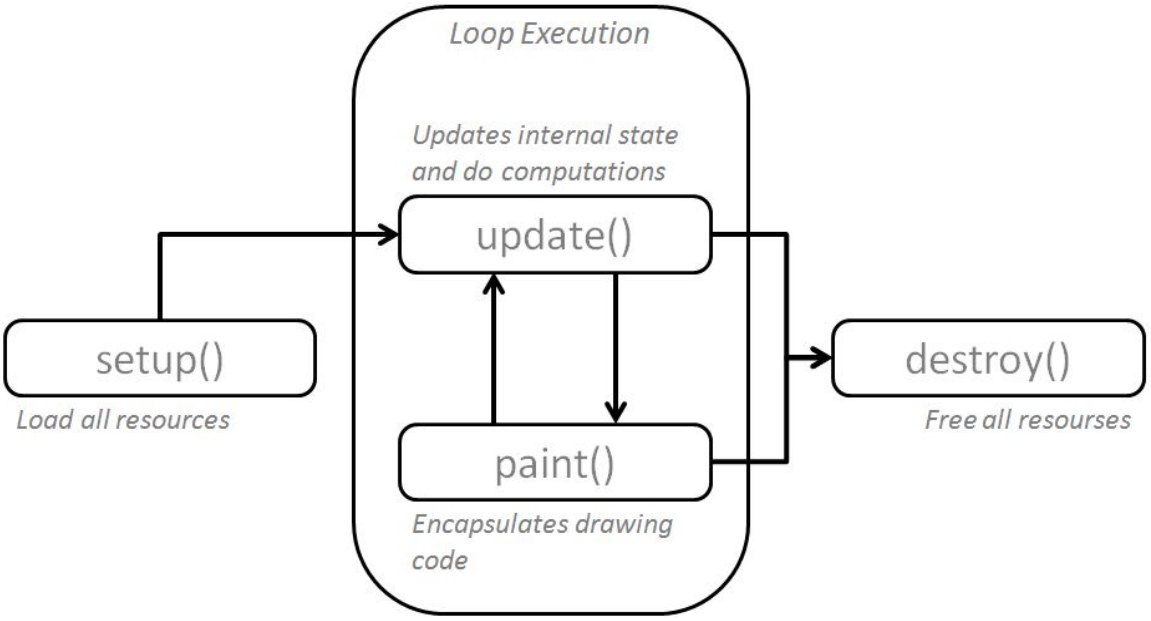

Figure 3. *Elements Life Cycle*

5.3. The Element Abstract Class

A customized element is implemented as a class which extends the abstract `Element` class. Some information about the element may be provided directly within the default constructor (name, description, version and author), or read from a configuration file within the package.

For example:

```
import scholz.dance.performance.element.Element;
import scholz.dance.performance.Performance;

public class MyElement extends Element {
    public MyElement() {
        super();
        super.setName("Purple Haze Element");
```

```
        super.setDescription("Purple Haze are in my brain.");
        super.setVersion("1.3.2");
        super.setAuthor("Jimi Hendrix");
    }
}
```

The methods below are not abstract, and have been implemented as empty methods in the `Element` class, so you are not supposed to override them if you do not need them to perform any particular operation.

```
public void setup() {}
public void update() {}
public void paint() {}
public void destroy() {}
```

After your element is implemented, you can add it to the performance, encapsulated within a Custom Element, through the `PerformanceFacade` method bellow:

```
public void addElement
            (CustomElement element, int track, int position, boolean shift);
```

Lower tracks will appear in front of higher tracks. Position indicates which column of the element's table (zero indexed) your element will be inserted. Finally, the shift flag tells the facade if your element will override the previous one, or shift it to the right on that layer.

5.4. Writing a Hello World Element

We'll write a simple "Hello World" element as an example. However, we'll use some features to make this example more illustrative. This "Hello World" paints a circle in the middle of the screen, which gets bigger with time and changes color every time it gets to occupy the whole screen, starting all over again, indefinitely.

See source code of "HelloWorld.java" in your *marine core project*. Run its `main()` method to see it in action. In the `main()` method, the `prepareExecution()` is called, then 5 seconds later, `play()` is called. The Hello World element runs for 20 seconds. Finally, `stop()` method is called, interrupting the element's execution.

Remember that if you want to run any performances from within your custom element plug-in project, you must follow the instructions on Section 5.1.2.

5.5. Defining Elements' Parameters

The Element abstract class has the following attribute:

```
protected HashMap<Integer, Parameter> parameters;
```

The hashmap provides a faster retrieval during element execution. However, you may read all element parameters within `setup()` method, and use local variables for them during the

whole element execution lifetime, for even faster execution. Each parameter must have an unique integer key, in the scope of the element. One approach is creating constants in your element for the keys, so your code looks cleaner when you access the parameters within the elements methods.

To add a parameter, usually in your element constructor, use:

```
super.parameters.put(Integer key, Parameter parameter);
```

To retrieve a parameter, within your element subclass, use:

```
Parameter myParameter = super.parameters.get(Integer key);
```

5.5.1. Parameter Subtypes

The `Parameter` class is abstract. For each parameter, you should use one of the following subclasses, and properly cast them when retrieving your parameter:

1. **`ColorParameter`:** holds a color value (ARGB) within an *int* variable; use `ColorUtil` class for methods to easily handle colors by R, G, B and alpha values;

2. **`FlagParameter`:** holds a *boolean* value;

3. **`NumberParameter`:** holds a *float* value, together with minimum and maximum values;

4. **`TextParameter`:** holds a String value;

5. **`TimeParameter`:** holds a *long* value, which is expected to be the time in milliseconds; use `TimeUtil` for methods to easily perform time operations;

6. **`SelectionParameter<T>`:** this is the more complex parameter, as it holds a list of possible values and a selected value, of generic type `T`; when creating this element, you should also provide a String value to indicate the attribute or the method that should be invoked on `T` to retrieve the label of the values (for UI purposes, for instance); also, as `T` is an unknown type at compilation time, you should provide an instance of `ParameterSerializer<T, I, O>`, where `I` is an input serializer (such as `DataInputStream`) and `O` is an output serializer (such as `DataOutputStream`); the current serialization implementation will expect these types.

5.5.2. Parameter Input Types

For UI purposes, each parameter holds an input type. Input types are defined in the `mustic.scholz.marine.core.performance.element.InputTypeEnum` enumerator and are one of the following, which map to the described expected UI element:

1. **FLOAT_SLIDER**: a slider from minimum value to maximum value, allowing values in the real numbers domain; should map to a `NumberParameter`;

2. **INTEGER_SLIDER**: a slider from minimum value to maximum value, allowing values in the integers domain; should map to a `NumberParameter`;

3. **TEXT_BOX**: a text box; should map to a `TextParameter`;

4. **FILE_PATH**: a text box which opens a window to locate files when clicked; should map to a `TextParameter`;

5. **SELECTION_LIST**: a list of values, which exhibits each value label; should map to a `SelectionParameter<T>`;

6. **SWITCH**: a switch, which allows true or false values; should map to a `FlagParameter`;

7. **TIME**: a time input (hour, minutes and seconds, or similar); should map to a `TimeParameter`;

8. **COLOR**: a color palette, which allows to chose one of several colors or setup a custom color providing R, G and B values; should map to `ColorParameter`.

5.6. Packing an Element as a Plug-in

Plug-in element projects should be mavenized, as in *marine template element project*. To pack your code as a plug-in, you just need to generate a JAR file, with the following characteristics:

1. Binary files must be accessible through the root, with the Java packages conventional folder structure;

2. The folder `config` must be included, with `plugin.config` file;

3. The folders `lib` and `res` may be included, but make sure you properly adjust `pom.xml` to avoid including unnecessary files;

The template `pom.xml` comply with all these characteristics, removing some unnecessary files (included in the project to allow test runs within plug-in element project, but unnecessary for plug-in packaging and deployment).

Therefore, in order to create a plug-in MAR file:

1. Right-click in the plug-in project, and select “Run as… >> Maven Clean”, to delete contents of the `target` directory;

2. Right-click in the plug-in project, and select “Run as… >> Maven Build”, fill “goal” field with “package” (without quotes), and press “run”;

The MAR file will be generated within the directory named `target`.

5.6.1. Element Icons

marine manages element icons automatically, so user interface implementations can easily find each element’s icon and large icon. When packaging your own element, make sure:

1. Under the “img” directory, you include an icon image, sized 36x36, and named icon.png;
2. Under the “img” directory, you include a large icon image, sized 78x78, and named icon-large.png;

6. Features and Skeleton Information

6.1. Reading Features

Movement features, as well as performer skeleton, are managed by the `FeaturesPool`. It is a Singleton class, which is asynchronously updated by the `IInputReceiver`.

Feature access occurs through two methods:

```
public Feature getFeature(String id);
public Feature getFeature(String id, JointEnum joint);
```

Calls to `getFeature(id, null)` are equivalent to calls to `getFeature(id)`. The embedded features’ identifiers are all declared as public constants in the `BuiltInFeatures` class. If a given feature is not present in the pool, both methods will return “null”.

If Kinect loses track of the performer, feature values will stop being updated. In this situation, the last valid values will be returned until new values overwrite them.

A list of all features in the pool may be retrieved by the method:

```
List<Feature> FeaturesPool.getAllFeatures();
```

However, if the input messages have not started to arrive yet, this method may return an empty or incomplete list.

The safer way to retrieve a list of all built-in features, with default values, is through:

```
List<Feature> FeatureFactory.buildAllFeatures();
```

6.2. Reading Skeleton Information

By now, *marine* deals with a single performer only. The first skeleton read by MS Kinect is modeled as a `PerformerPosition` instance. Performer position access occurs through the method bellow, from `FeaturesPool`:

```
public PerformerPosition getPerformerPosition();
```

If there is no performer position detected, the method will return “null”. From the performer position, it is possible to access all joint positions, through:

```
PerformerPosition.getPositions();
PerformerPosition.getPosition(JointEnum joint);
```

Since `FeaturesPool` is a Singleton, the complete statement to retrieve a performer position is:

```
PerformerPosition performer =
                  FeaturesPool.getInstance().getPerformerPosition();
```

It is possible to accept performer positions read some time before as valid, by calling:

```
PerformerPosition performer =
                  FeaturesPool.getInstance()
                  .getPerformerPosition(long expirationMillisecs);
```

Every performer position has a timestamp and an array of `JointPositions`, containing a joint (`JointEnum` attribute) and a position vector. Each position vector has three dimensional coordinates (x, y, z.).

The `SkeletonAgent` paints the performer skeleton on screen, by reading every joint position and creating corresponding “bones” between joints. Try executing “`SkeletonAgentTest.java`” to see it in action.

6.3. Writing Custom Feature Extractors

Although future versions may allow the use of a sequence of performer positions up to some limit, currently, custom features can only be based on the most up to date `PerformerPosition` information.

Custom features are computed by feature extractors. Feature extractors must implement the `FeatureExtractor` interface. Then, they must be registered in the `MovementManager`, to be updated everytime the performer position is updated. The `FeatureExtractor` interface has a single method to be implemented:

```
public Feature computeFeature(PerformerPosition performerPosition);
```

In order to register a feature extractor, retrieve the `MovementManager` instance and call:

```
public void addFeatureExtractor(FeatureExtractor extractor);
```

7. Coordinates Systems and Calibration

7.1. Introduction

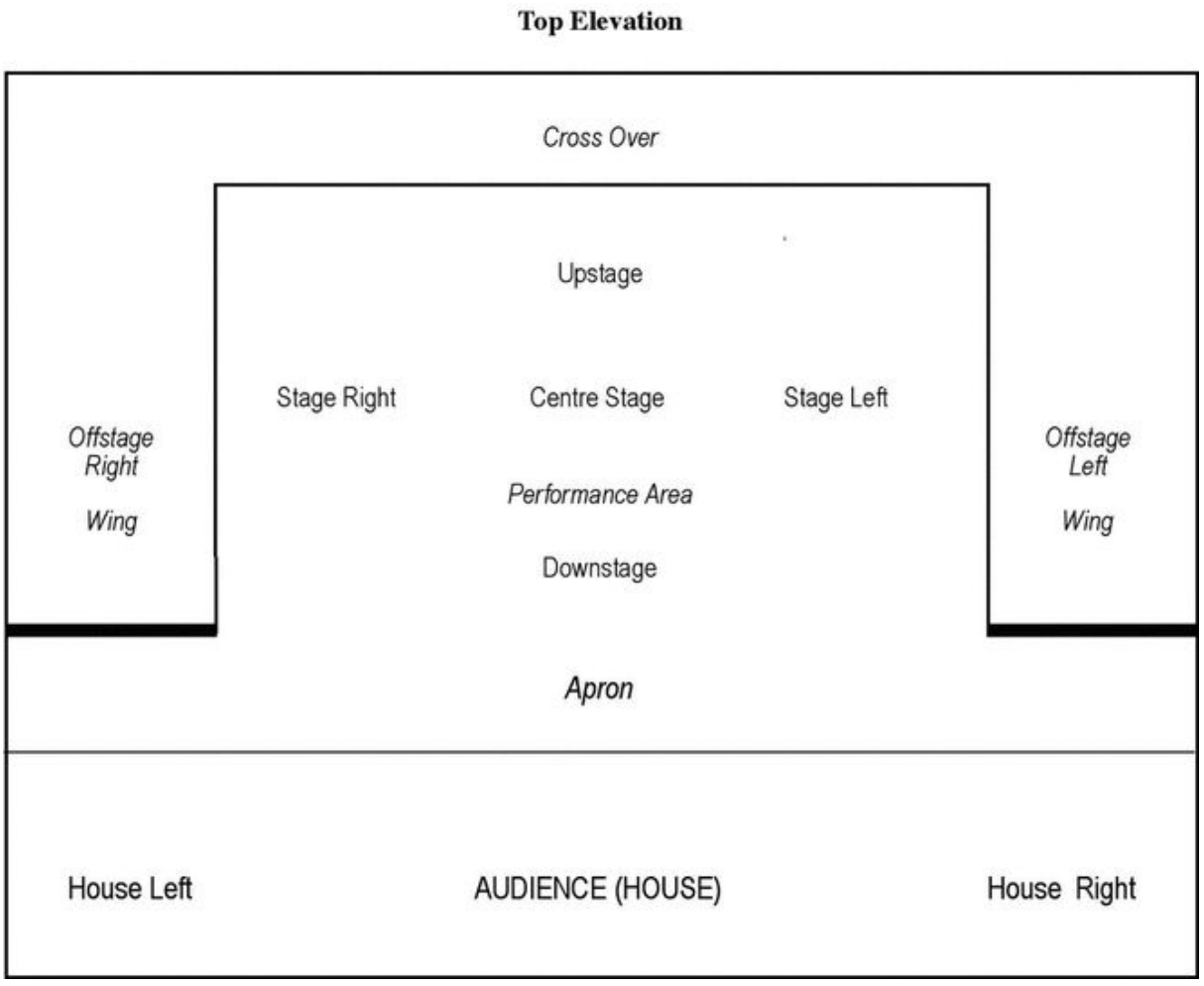

Figure 4. *Italian stage parts.*

There are different coordinates' systems (CS for short) going on, which need to be matched. The first one is the input device CS, which may vary from device to device. MS Kinect uses "right handed" coordinates: its (0,0,0) point in the focal point of the infrared camera, with x-axis growing to the left of the device (point your right thumb on that direction), the y-axis growing upwards (point your right indicator finger on that direction), and the z-axis growing forward (point your right middle finger on that direction).

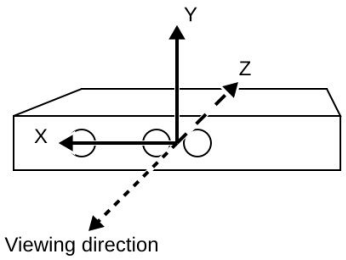

Figure 5. *Coordinates system of the Microsoft Kinect.*

marine coordinates' system (CS, for short) differs from Processing coordinates system. Stage based CS has been chosen so that all graphic interactions can be programed more intuitively. The reference/stage CS has its (0, 0, 0) point in the middle of the stage, with x-axis growing to the left of the stage (if you are in the audience, left of the stage is your right side), y-axis growing upwards and z-axis growing in the direction of downstage. Though, it is a "left hand" CS (left thumb pointing x-axis, and so on). The stage CS is considered the reference CS. All skeleton points will be translated to that CS. The figure below shows the coordinates used.

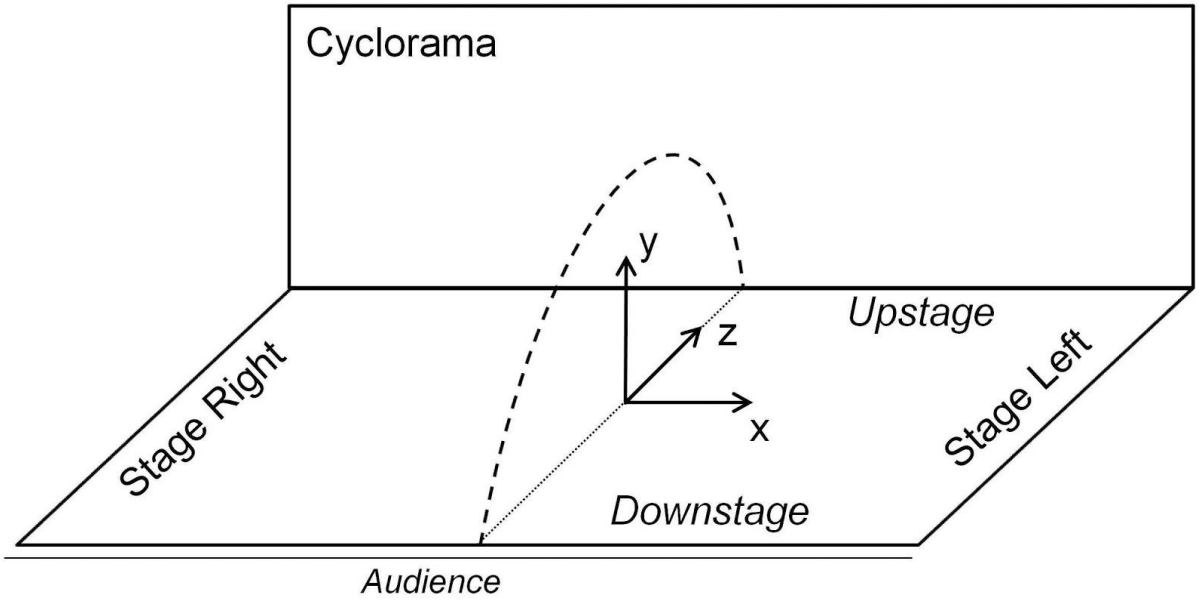

Figure 6. *Marine coordinates' system.*

Finally, Processing 3+ has its own CS, with (0, 0, 0) point in the top-left corner of the screen, with x-axis growing to the right of the screen, y-axis growing downwards and z-axis growing to the front of the screen (also a "left hand" CS, but with a 180° rotation over the x-axis, comparing to the reference CS).

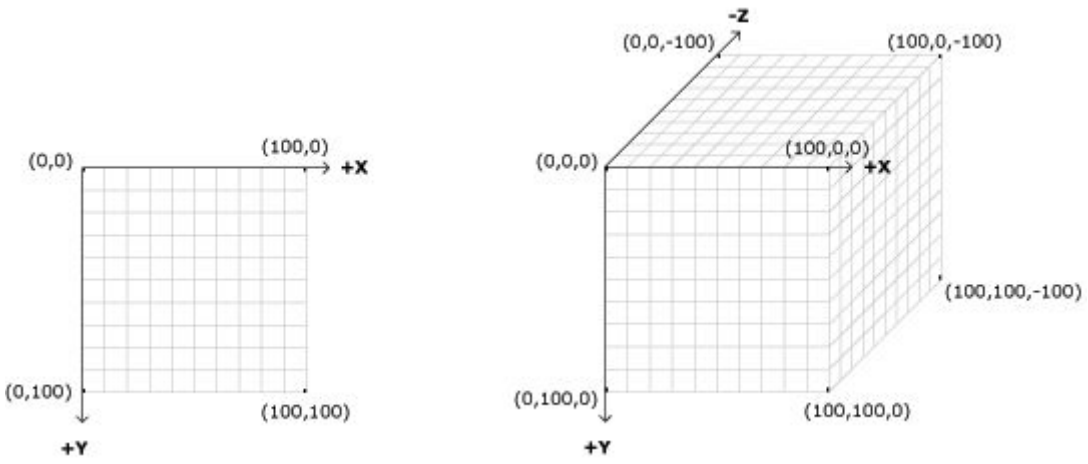

Figure 7. *Processing 3+ coordinates' system.*

7.2. Input Device Angles

The input device is considered to be positioned from the first row of the audience, centralized with the stage, parallel to the floor plane, pointing to the upstage, until 180° in the upstage (turned upside down), including a 90° position, in the ceiling, just above the (0, 0, 0) point, pointing towards the floor. Be aware that the use of the MS Kinect pointing to the audience may cause confusion if the audience is too close to the stage (within the recognition area). Using the input device in other positions will require the respective adjustments in the projectors to match.

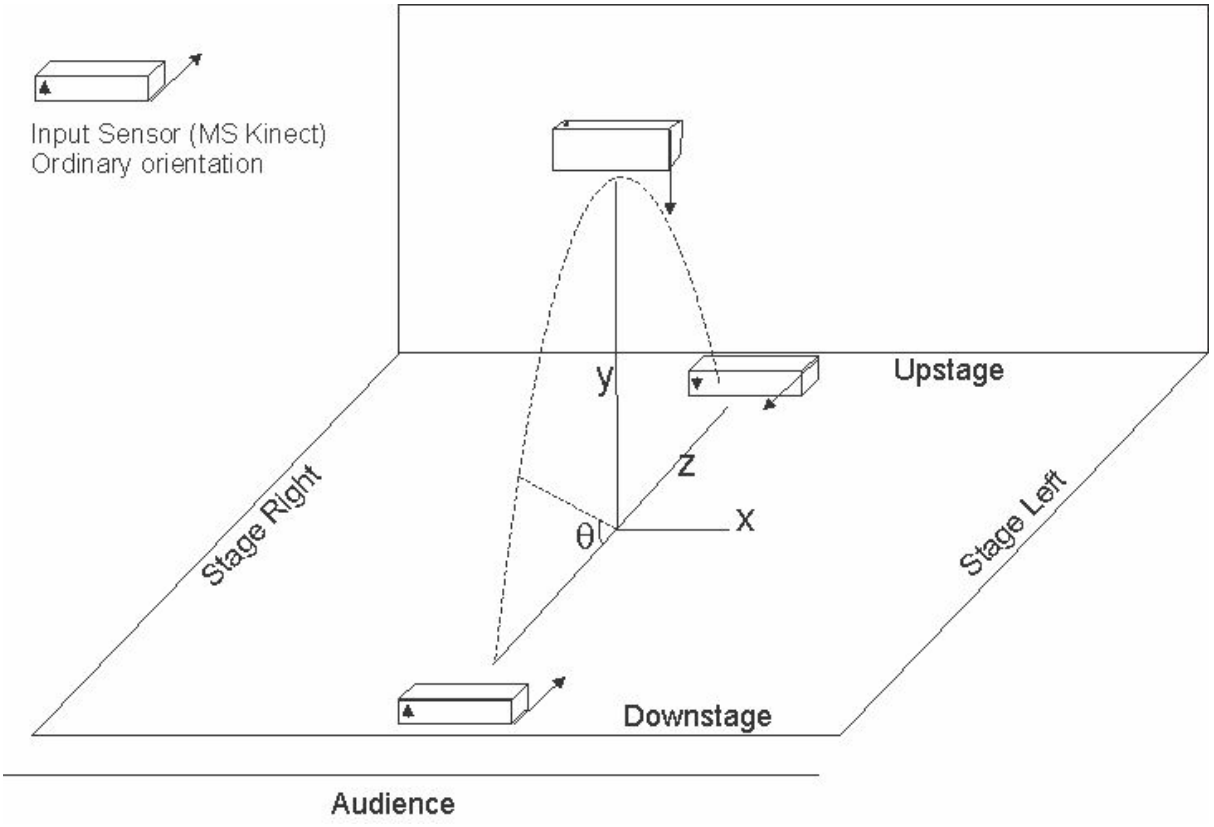

Figure 8. *Input device possible arrangements.*

7.3. Projection Modes

There are two projection modes. Choosing a projection mode will change the camera position, so that what is projected matches with the real world objects in the stage.

It is important to highlight that projection modes are independent of input device position. *marine* will compute the coordinates accordingly when you define the input device angle (theta) and projection mode desired when calibrating your stage arrangement. See Section 8.5.

7.3.1. Floor Mode

In this mode, the projector is expected to be pointed to the floor, on a 90° angle, so that real world coordinates match projected coordinates. Depending on your stage constraints, you may have to use the projector in a different angle, slopingly projecting in the floor, and adjust a compensation through projector's angle setup configurations.

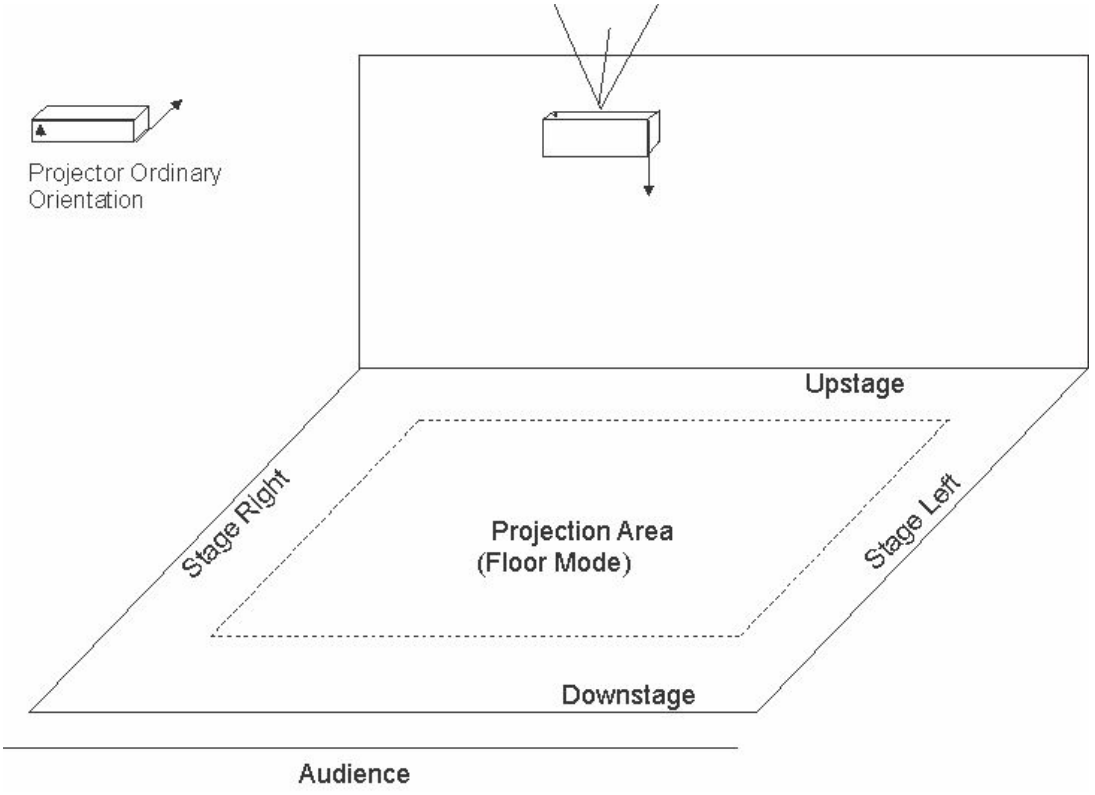

Figure 9. *Floor projection scheme.*

7.3.2. Wall Mode

In wall mode, the projector is expected to be pointing to a screen, parallel to an imaginary wall in the upstage, so the real world coordinates match projected coordinates.

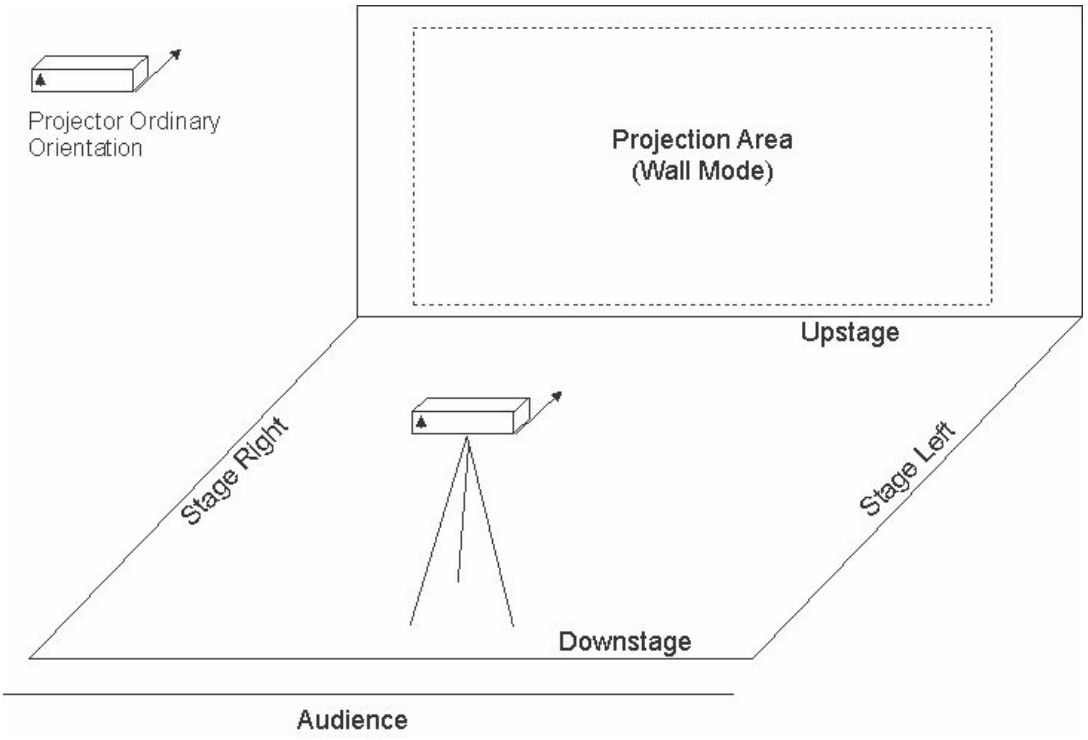

Figure 10. *Wall projection scheme.*

7.4. Using 2D Screen Based Coordinates

It is possible to paint 2D objects, regardless of the projection mode being used, with screen based coordinates. In this mode, point (0,0) will be in the center of the screen, with x-axis growing to the right and y-axis growing upwards. To activate 2D mode painting, call `performance.begin2D()` before your painting code. When you finish painting, call `performance.end2D()`. These calls must come always in pairs, otherwise unpredictable errors may occur, including with other elements executing in parallel, as the performance instance is shared.

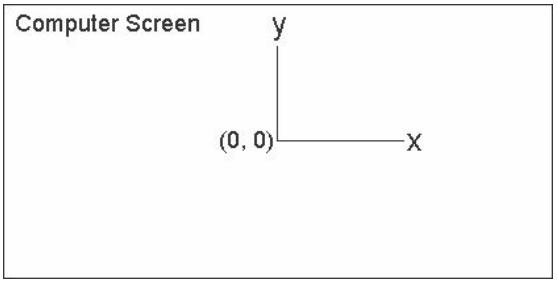

Figure 11. *Bidimensional screen based coordinates' system.*

A known **drawback** of this approach is that Processing text drawing appears upside down. Whenever a text is to be drawn, the adjustments must be done by the programmer.

7.5. Calibrating Camera and Coordinates

Some agents which use performer position to print over might be very sensitive to the exact position of the sensor and the projector, as well as the measures used in the coordinates

systems. Therefore, an element has been developed to help on the task of calibrating the scene, once the sensor and the projector are well positioned. Of course, a given calibration setup may be saved, as well as a previous calibration setup may be loaded.

7.5.1. Loading a Previous Calibration and Saving Current Calibration

The `PerformanceFacade` provides two methods for loading from file system and saving to file system a given calibration:

```
public void loadCalibrationFile(String path);

public void saveCurrentCalibrationToFile(String path);
```

7.5.2. Calibration File

The calibration file is a text file in which the calibration properties are saved. Each line of this file contains a pair of key and value, in the format “key=value”.

The default calibration settings will generate a file such as the example below:

```
input.device.translation.z=-3.6999996
input.device.translation.y=0.9000001
input.device.translation.x=0.0
projection.plane=floor
projection.perspective=true
input.device.pointsperunitratio=220.0
input.device.angle=0.0
camera.distance=430.0
camera.angle=0.5235988
```

The properties keys are saved as constants in the `CalibrationPropertiesManager` class. All properties are numeric, except for:

1. “pojection.plane”: this property accepts “wall” or “floor” as valid values; if any other value is used, it will consider the projection plane “wall”;

2. “projection.perspective”: this property accepts “true” or “false” as valid values; if any other value is used, it will consider the projection is using perspective (“true”).

7.5.3. Calibration Element

In order to allow easy calibration, an element has been developed for that purpose. The element paints some markers on screen and keyboard may be used to adjust values.

Run the class “`mustic.scholz.marine.test.Calibrator.java`”, and use the commands printed on screen to change settings.

The markers are:

1. **Axes**: a red X axis, a green Y axis and a blue Z axis, as well as a grey box are drawn; this is intended to help understanding the axis positions in real world;

2. **Unit Line**: draws an orange line, one unit long, parallel to X axis, at z = -0.5 units, centralized with Z axis (for MS Kinect, 1 unit = 1 meter); this is intended to help adjusting “pixels per unit ratio” setting;

3. **Perspective Boxes**: transparent boxes on each axis are drawn in order to allow visually identification of perspective mode; the use of perspective make further objects appear smaller than closer objects; when perspective is not used, the boxes look like squares, as their further planes appear the same size as the closer planes and are overwritten;

4. **Skeleton**: recognized skeleton is drawn on screen, so that users can see the performer position on screen coordinates;

5. **Calibration Info and Commands**: current calibration information and keyboard commands are drawn in the left corner of the screen.

8. Input Listeners

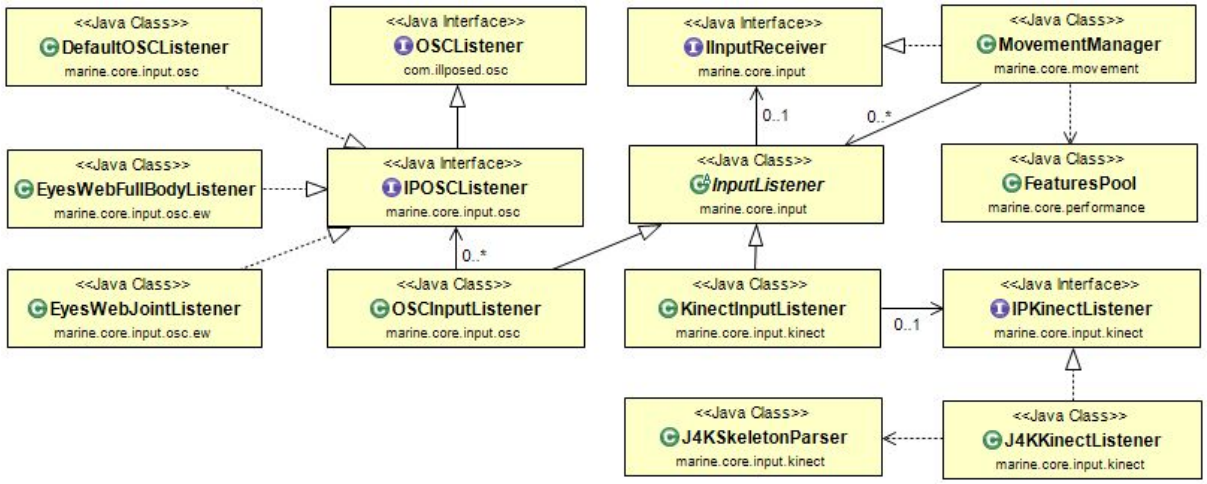

Figure 12. *marine Input Listeners Model (“mustic.scholz” omitted on package names)*

Input listeners are responsible for the translation between a given input stream and *marine* performer and feature information. They may be used to read performer position from different movement tracking hardware, as well as to read features computed on different softwares.

An input listeners must extend the `InputListener` abstract class, which implements `Runnable` and contains a reference to the object which will be called back when an event occurs. As for organization of code, a single input listener is allowed to encapsulate other

“sub-listeners”, which are plain Java classes in which the programmer may want to encapsulate different input specific logic or distinct implementations. Though, two methods must be overwritten:

```
public abstract void addListener(Object listener);

public abstract void removeListener(Object listener);
```

The method called to set up and start the listener must also be overwritten:

```
public abstract void startListening();
```

When performer position or features are received and decoded as such (i.e., new `Feature` objects or `PerformerPosition` objects have been properly created from the input data read), the input listener must perform a callback, by calling one of these methods, from its superclass `InputListener`, accordingly:

```
public void receivePositionMessage(PerformerPosition position);

public void receiveFeatureMessage(List<Feature> features);
```

These calls will allow the `FeaturesPool` to be properly updated. For sample implementations, please check out the following packages:

```
scholz.dance.core.input
scholz.dance.core.input.kinect
scholz.dance.core.input.osc
```

9. Advanced Features

9.1. Painting on Screen

Currently, every element has direct access to the `PApplet` and uses it to paint on screen. This approach, however, is not safe, as plug-in elements must push matrix transformations to the main graphics (without popping them), what may affect other elements.

For more details on how to paint into a `PApplet`, `PImage` or `PGraphics`, see Processing 3+ documentation at http://processing.org or check Javadoc at: http://processing.github.io/processing-javadocs/core/.

The `PApplet` instance is available to the elements through the variable "`performance`", in the superclass `Element`.

9.2. Sending OSC Messages

OSC messages can be easily sent using `OSCOutputManager`. It is possible to add messages to a buffer, and send them at once. After creating an instance of `OSCOutputManager`, set the target host and port by calling:

```
public void setTarget(String host, Integer port);
```

Then, enqueue messages by calling:

```
public void addSender(OSCOutputMessage sender);
```

Remember that `OSCOutputMessage` is an interface, implemented by `DefaultOSCMessage`, but you may need to build your own messages.

To send the messages enqueued, use:

```
public void sendMessages();
```

Sending messages remove them from the queue. However enqueued messages may be also removed by calling:

```
public void removeSenders();
```

The OSC Output Manager implements `Runnable`, so it can run asynchronously, if needed. In this case, you only need to enqueue messages, so it will send them and empty queue from times to times. You may want to control the frequency of the output, by setting the sleep time (in milliseconds) between each batch delivery:

```
public void setSleepInterval(long interval);
```

9.3. Sending MIDI Messages

MIDI messages can be easily sent using `MidiOutputManager`. It is possible to add messages to a buffer, and send them at once. After creating an instance of `MIDIOutputManager`, start enqueuing messages, by calling:

```
public void addShortMessage(int status, int channel, int data1, int data2);
```

To send the messages enqueued, use:

```
public void sendMessages();
```

Sending messages remove them from the queue. However enqueued messages may be also removed by calling:

```
public void removeMessages();
```

The MIDI Output Manager implements `Runnable`, so it can run asynchronously, if needed. In this case, you only need to enqueue messages, so it will send them and empty queue from times to times. You may want to control the frequency of the output, by setting the sleep time (in milliseconds) between each batch delivery:

```
public void setSleepInterval(long interval);
```

9.4. Sending DMX Messages

DMX messages can be easily sent using `OpenDMXOutputManager`. To create an instance of `OpenDMXOutputManager`, execute the following line, indicating which universe (>= 0) is binded to this manager, how many channels you are using (starting on channel 1) and how many milliseconds the system should wait before sending two consecutive messages (for performance optimization):

```
OpenDMXOutputManager manager
      = OpenDMXOutputManager.init(universe, channels, sleepInterval);
```

You can change the universe and amount of channels, by calling:

```
public void setUniverse(int universe);
public void setChannels(int channels);
```

Then, you must write data into each channel, individually:

```
public void setData(int channel, int data);
```

When a DMX message is sent, it goes with the current values in the `data` array.

The DMX Output Manager implements `Runnable`, so it can run asynchronously, if needed. In this case, you only need to keep `data` array up to date.

10. Bugs and Future Improvements

If you find a bug or if you want to suggest a new features or improvement, please, drop me a line at:

contact@marineframework.org

Or create a ticket at:

https://bitbucket.org/ricardoscholz/marine

ANEX I - Third Party Softwares Terms of Use

The following terms of use may be out of date at the time you read this document. Please, make sure you read the most up to date copy at the respective providers official websites.

I.A - EyesWeb Licence Agreement

Original EyesWeb Licence Agreement may be found at: http://www.infomus.org/eyesweb_license_ita.php

The following is a transcription of the EyesWeb Licence Agreement, as of december 15th 2016:

EYESWEB CAN BE FREELY DOWNLOADED BUT PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE

Use of EyesWeb (hereinafter 'SOFTWARE') is contingent on your agreement to the following terms:

WARRANTY & USE: DIST

University of Genoa grants you a limited, non-exclusive license to use the SOFTWARE free of charge for ANY purpose, commercial or private, without restrictions. DIST - University of Genoa makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. DIST - University of Genoa is not obligated to provide maintenance or updates for the SOFTWARE.

DISTRIBUTION

the SOFTWARE, in original version or in part, may be freely distributed, provided that this copyright and permission notice appear on all copies and supporting documentation, and that the DIST - University of Genoa copyright notices are referred in the following ways:

- DIST - University of Genoa's copyright notice should be included in the documentation, regardless of the media used to supply the documentation;
- The DIST - University of Genoa and EyesWeb Logos have to appear on software packages based on EyesWeb or using it, and on any related promotional material (DIST - University of Genoa makes the logos available on the EyesWeb ftp site ftp://ftp.infomus.org);
- in the 'about box' of the product, in the case that it is not the EyesWeb about box, DIST - University of Genoa and EyesWeb must be cited in the following manner: EyesWeb is copyright (c) Laboratorio di Infomatica Musicale - DIST - University of Genoa (http://infomus.dist.unige.it);
- any public use of EyesWeb, or the distribution of any application based on EyesWeb, must be preliminarily notified to info@infomus.org;
- this license must be notified to any third party to which EyesWeb is redistributed.

This license only covers EyesWeb and the libraries provided in the original installer. Other extensions (libraries or patches), provided by DIST or by third parties, may be subject to any license, provided that it is not in contrast with this license.

I.B - J4K Terms and Conditions

Original J4K Terms and Conditions may be found at: http://research.dwi.ufl.edu/ufdw/terms.html

The following is a transcription of the J4K Terms and Conditions, as of december 15th 2016:

THIS SOFTWARE IS PROVIDED TO YOU "AS IS," AND WE MAKE NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER WITH RESPECT TO ITS FUNCTIONALITY, OPERABILITY, OR USE, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR INFRINGEMENT. WE EXPRESSLY DISCLAIM ANY LIABILITY WHATSOEVER FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST REVENUES, LOST PROFITS, LOSSES RESULTING FROM BUSINESS INTERRUPTION OR LOSS OF DATA, REGARDLESS OF THE FORM OF ACTION OR LEGAL THEORY UNDER WHICH THE LIABILITY MAY BE ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY OR LIKELIHOOD OF SUCH DAMAGES.

I.C - Processing Copyright Notice

Original Processing.org Copyright Notice may be found at: https://www.processing.org/copyright.html

The following is a transcription of the Processing Copyright Notice, as of december 15th 2016:

Processing was started in Spring 2001 by Ben Fry and Casey Reas. Fry was a PhD candidate at the MIT Media Laboratory and Reas was an Associate Professor at the Interaction Design Institute Ivrea. While Fry and Reas were employees of these institutions, Processing began as a personal initiative and development took place during the night and weekends through 2003. MIT indirectly funded Processing through Fry's graduate stipend and Ivrea indirectly funded Processing through Reas's salary. Due to his research agreement with MIT, all code written by Fry during this time is copyright MIT.
In summer 2003, Ivrea funded four individuals to work on the project for a few months. This resulted in Dan Mosedale's preprocessor using Antlr and Sami Arola's contributions to the graphics engine. The code for these elements are both copyright 2003 Interaction Design Institute Ivrea.
In August 2003, Reas left the Interaction Design Institute Ivrea and in June 2004, Fry left the MIT Media Laboratory. The code and complete reference written since June 2004 are copyright Ben Fry and Casey Reas.
Portions of the code were written by other contributors and are attributed in the source code. For example, portions of the graphics engine were written by Karsten Schmidt. There are many contributions to the Exhibition and Examples on the Processing.org website and these are attributed in context.
The Reference for the Language and Environment are under a Creative Commons license which makes it possible to re-use this content for non-commercial purposes if it is credited.