

# PixelLight Script Documentation



February 23, 2012  
PixelLight 0.9.11-R1



The content of this PixelLight document is published under the Creative Commons  
Attribution-NonCommercial-ShareAlike 3.0 Unported  
Copyright © 2002-2012 by The PixelLight Team



# Contents

<b>1. Introduction</b>	<b>7</b>
1.1. Interactive Realtime Run-Time Type Information (RTTI) Browsing Graphical User Interface (GUI) . . . . .	7
1.2. External Dependences . . . . .	8
<b>2. Script</b>	<b>9</b>
2.1. Namespaces . . . . .	10
2.2. Global Variables . . . . .	11
2.3. Global Functions . . . . .	12
2.4. Object-Oriented Programming (OOP) . . . . .	13
2.5. RTTI Objects . . . . .	16
2.5.1. Properties . . . . .	17
2.5.2. Attributes . . . . .	17
2.5.3. Methods . . . . .	18
2.5.4. Slots . . . . .	19
2.5.5. Signals . . . . .	20
2.5.6. Creating a RTTI Class Instance via Script . . . . .	22
<b>3. Script Backends</b>	<b>23</b>
<b>4. Script Bindings Plugin</b>	<b>25</b>
4.1. PL . . . . .	25
4.2. PL.System . . . . .	25
4.3. PL.Log . . . . .	25
4.4. PL.System.Console . . . . .	26
4.5. PL.Timing . . . . .	26
4.6. PL.ClassManager . . . . .	27
<b>5. PLCore RTTI Classes</b>	<b>29</b>
5.1. PLCore::Object Class . . . . .	29
5.1.1. Methods . . . . .	29
5.1.2. Signals . . . . .	30
5.2. PLCore::CoreApplication Class . . . . .	30
5.2.1. Methods . . . . .	30
5.3. PLCore::FrontendApplication Class . . . . .	30
5.3.1. Methods . . . . .	30

5.4.	PLCore::Frontend Class . . . . .	30
5.4.1.	Methods . . . . .	30
5.5.	PLCore::ApplicationContext Class . . . . .	32
5.5.1.	Methods . . . . .	32
<b>6.</b>	<b>PLInput RTTI Classes</b>	<b>35</b>
6.1.	PLInput::Controller Class . . . . .	35
6.1.1.	Attributes . . . . .	35
6.1.2.	Signals . . . . .	35
6.2.	PLInput::Control Class . . . . .	35
6.2.1.	Methods . . . . .	35
6.3.	PLInput::Axis Class . . . . .	36
6.3.1.	Methods . . . . .	36
6.4.	PLInput::Button Class . . . . .	36
6.4.1.	Methods . . . . .	36
6.5.	PLInput::Effect Class . . . . .	37
6.5.1.	Methods . . . . .	37
6.6.	PLInput::LED Class . . . . .	37
6.6.1.	Methods . . . . .	37
<b>7.</b>	<b>PLRenderer RTTI Classes</b>	<b>39</b>
7.1.	PLRenderer::RendererApplication Class . . . . .	39
7.1.1.	Methods . . . . .	39
<b>8.</b>	<b>PLScene RTTI Classes</b>	<b>41</b>
8.1.	PLScene::SceneApplication Class . . . . .	41
8.1.1.	Methods . . . . .	41
8.2.	PLScene::SceneNode Class . . . . .	41
8.2.1.	Attributes . . . . .	41
8.2.2.	Methods . . . . .	42
8.2.3.	Signals . . . . .	45
8.3.	PLScene::SceneNodeModifier Class . . . . .	45
8.3.1.	Attributes . . . . .	45
8.3.2.	Methods . . . . .	46
8.4.	PLScene::SceneContainer Class . . . . .	46
8.4.1.	Attributes . . . . .	46
8.4.2.	Methods . . . . .	47
8.4.3.	Signals . . . . .	48
<b>9.</b>	<b>PLEngine RTTI Classes</b>	<b>49</b>
9.1.	PLEngine::EngineApplication Class . . . . .	49
9.1.1.	Methods . . . . .	49
9.1.2.	Signals . . . . .	50

9.2. PLEngine::ScriptApplication Class . . . . .	50
9.2.1. Attributes . . . . .	50
9.2.2. Methods . . . . .	50
<b>10. PLGui RTTI Classes</b>	<b>53</b>
10.1. PLGui::Gui Class . . . . .	53
10.1.1. Methods . . . . .	53
10.2. PLGui::Widget Class . . . . .	53
10.2.1. Attributes . . . . .	53
10.2.2. Methods . . . . .	53
10.2.3. Signals . . . . .	54
10.3. PLGui::GuiApplication Class . . . . .	56
10.3.1. Methods . . . . .	56
<b>11. Contact</b>	<b>57</b>
<b>A. Virtual Standard Controller</b>	<b>59</b>
<b>B. Input Examples</b>	<b>65</b>
<b>C. Scene Examples</b>	<b>67</b>
C.1. Constructing Scenes . . . . .	67
C.2. Loading Scenes . . . . .	68
<b>Abbreviations</b>	<b>71</b>



# 1. Introduction

**Target Audience** This document is meant for programmers and artists capable of doing some scripting.

**Motivation** This script documentation is intended for script programmers. While this document also talks about some quite basic stuff, the targeted audience has already some fundamental programming experience. If you're totally new to the field of programming in general, it's highly recommended to read some beginner literature first <sup>1</sup>.

The reason for creating the script interface was to have a minimalistic and universal interface to work with script languages. Instead of inventing an own script language, we use a backend design pattern to use already available common script languages like *Lua*, *JavaScript*, *Python* or *AngelScript*.

Why should you care about scripting in the first place when PixelLight is written in C++ and you have the whole C++ Application Programming Interface (API) and even the complete source code of the project at our hands? Well, let me say it this way: Not every programmer is a programmer. What's meant by this statement is, that within development teams not every team member is able to or want's to work in C++. When it comes to implement application logic like

”When the user clicks on this door then let an anvil fall down on him”

, there's no real need to let everything be written by experienced (and therefore usually expensive) software developers within C++. Let's say a graphics artist has finished his work and has enough of colourful things for a few hours - why shouldn't he be allowed to implement some simple application logic? By using scripting, it's usually no big deal to let other persons than experienced software developers do some coding.

## 1.1. Interactive Realtime RTTI Browsing GUI

The script system heavily relies on the PixelLight RTTI system. So, you probably may want to know which classes, attributes and so on are available for usage. Please note that the purpose of this document is not to describe each and every possible global function, RTTI classes, methods, attributes and so on. Due to the highly plugin driven architecture of PixelLight, this would be impossible anyway. There are several chapters about fundamental RTTI classes, but that's just a tiny portion of what's available.

---

<sup>1</sup>For example *Programming in Lua (first edition)* which is online available at <http://www.lua.org/pil/> if you're just interested in scripting using Lua

## 1. Introduction

For the rest, please refer to the interactive realtime RTTI browsing GUI which can be accessed by using for instance the tool *PLViewerQt* which comes with the Software Development Kit (SDK). This way, the *documentation* is automatically always up-to-date and also contains your own, custom extensions.

## 1.2. External Dependences

There's no individual *PLScript* project, it's part of *PLCore*. The **PLCore** library only depends on *zlib* and *libpcre*, both are statically linked in.

## 2. Script

Because Lua is shipped with the official PixelLight SDK, this script language will be used within the examples. Please note that the PixelLight script API is script language independent. In fact, there's not even such a PixelLight script API. Everything that is connected to the RTTI of PixelLight can be accessed through script languages as long as the script backend has support for RTTI objects.

Within the PLCore documentation, there's a lot of inside detail information on how the RTTI works. The script support itself is implemented within PLCore because scripting is heavily using PLCore features like the RTTI. Therefore, adding script bindings or using RTTI objects within scripts is fairly straightforward and doesn't require the writing of thousands of proxy/wrapper classes exposing C++ functionality to script languages.

Certain non-RTTI parts of PixelLight are exposed to script languages through the loose plugin *PLScriptBindings*. Please note that within PixelLight, scripting is completely optional, not mandatory - unlike some other engines out there were one is only able to use scripting. So, you can use scripting, but you are not forced to do so.

In general, the abstract script interface of PixelLight supports the following script features:

- Global variables (with namespace support)
- Global functions, C++ calls script and script calls C++ (with namespace support)
- RTTI objects

As you will see when reading this document and playing around with the script support, the access to RTTI objects makes the script support quite powerful and universal. You're able to access properties (constant RTTI class information), attributes (variables within an object), methods (functions within an object), signals (events within an object) and slots (event handlers within an object).

Supported primitive data types are:

```
bool, float, double, int8, int16, int32, int64, uint8, uint16, uint32, uint64,  
PLCore::Object*, PLCore::Object&
```

Please note that not each script language/API may make such a detailed data type distinction. Because strings are fundamental within scripts, `PLCore::String` is supported as well. But if you're a 100% script programmer, without writing own new C++ components, the mentioned information is probably already too technically.

In general, the script support of PixelLight can be subdivided into the two following use-cases:

## 2. Script

- A C++ component is using an own script instance in a highly specialized way. The script scene node modifier is a good example for this. Such a script scene node modifier is attached to a scene node and is adding logic through a script.
- A scripted stand-alone application, meaning that the complete application logic is implemented within, for instance Lua, and can be executed by *PLViewer* which comes with the PixelLight SDK or custom applications. This doesn't mean that it's 100% script only because the more complex stuff will probably be done by used C++ components, but the wires are tied up by a script.

### 2.1. Namespaces

When a project is growing, one may run out of decent names or name conflicts happen by accident more often. Even if you're just starting a project and don't think that you'll end up with much source code, you should consider to use namespaces right from the beginning. That's the reason why this section comes even before introducing, for instance, global variables.

Most modern high-level computer languages support the concept of namespaces which, when used in the proper way, can help a developer with the naming task. An even bigger benefit in using namespaces than reducing name conflicts is, that elements are grouped together. An editor may, for example, just look for global variables inside a namespace called *PublicVariables* in order to make them visible and editable through a GUI - while other internal script variables will be hidden from the editor user. This way, it's not required to *extend* already existing, maybe even normed script languages like JavaScript (ECMA-262), by own constructs like a *public*-modifier to denote that a script variable should be accessible to the outside world. Please note that this was just an example, within the PixelLight core there's nothing like *PublicVariables* fixed build in.

While, on the first look, several script languages have no namespace support, they have so on the second look or it can be at least emulated. Let's take Lua as an example. Lua is a really compact language and many important features are missing within the core of the language... but on a deeper look they can be realized by using Lua's powerful and universal table concept. This is also true for namespaces.

When searching the internet for information how to do namespaces in Lua, one often finds examples like seen in source code 2.1. By writing *PublicVariables = {}*, a new global variable named *PublicVariables* is created and an empty table *{}* is assigned to it. The name *PublicVariables* is now addressing a Lua table. By writing *PublicVariables.Health = 42* the table *PublicVariables* gets a new entry with the key *Health* and the value *42*.

Personally, I don't like the style used in source code 2.1, it looks complicated to me. The result of source code 2.2 is the same, but in my personal opinion it's more readable and reminds me at e.g. C++.

When writing a script, it's probably a good idea to use such a *public*-namespace to mark that those script content should be visible to e.g. an editor while other stuff is for internal purposes only. The Lua script language does also have the concept of local

```

1 -- Create the "PublicVariables"-namespace
2 PublicVariables = {}
3 -- Create the variable "Health" within
4 -- the "PublicVariables"-namespace
5 PublicVariables.Health = 42
6 -- Create the variable "Gold" within
7 -- the "PublicVariables"-namespace
8 PublicVariables.Gold = 21

```

Listing 2.1: Global variables within a namespace (1)

```

1 -- Create the "PublicVariables"-namespace
2 PublicVariables = {
3   -- Create the variable "Health" within
4   -- the "PublicVariables"-namespace
5   Health = 42,
6   -- Create the variable "Gold" within
7   -- the "PublicVariables"-namespace
8   Gold = 21,
9 }

```

Listing 2.2: Global variables within a namespace (2)

variables by writing the keyword *local* in front of the variable declaration (see section 2.2 for details). Local variables are invisible to the C++ host, so, they can also be used to realize script internal variables.

I hope you got the idea why using namespaces, even in scripts, is a good and handy thing.

## 2.2. Global Variables

There's nothing fancy about global variables, so this section is quite compact. When using Lua, just assign somewhere within your script, as seen in source code 2.3, a value to a name and et voila, you've created a new global variable<sup>1</sup>.

Lua does also have the concept of local variables by writing the keyword *local* in front of the variable declaration as seen within source code 2.4. From the C++ side, it's possible to interact with global variables while local variables can not be seen. So, while the C++ side can access your global variable *Health*, it can't access your local variable *Gold*.

---

<sup>1</sup>Lua is loosely typed, the type of a variable depends on the value assigned to it and can be changed every time by just assigning another value with another type to it

## 2. Script

```
1 -- Create the global variable "Health"
2 Health = 42
3 -- Create the global variable "Gold"
4 Gold = 21
```

Listing 2.3: Global variables

```
1 -- Create the global variable "Health"
2 Health = 42
3 -- Create the local variable "Gold"
4 local Gold = 21
```

Listing 2.4: Local variables

**Global Variables added by the C++ Host** The C++ side is also able to create new, or to delete existing global variables. Whether or not this is in general useful is up to you. PixelLight components using scripts dynamically add a global variable named *this* pointing back to the script calling RTTI object. If you want to know more about the global *this* variable, have a look into section 2.5.

## 2.3. Global Functions

As soon as your script grows over a hand full of source code lines, the need for functions arises. Within Lua, global functions look like as seen in source code 2.5. The global

```
1 function MyFunction()
2     -- ... do some fancy stuff...
3 end
```

Listing 2.5: Global functions

function *MyFunction* can then be called by writing *MyFunction()*.

From the C++ side, it's possible to interact with global functions. Such global functions are usually used as script entry points and typical names for such global functions used as entry points are *OnInit*, *OnUpdate* and *OnDeInit* which are called after as script has been loaded, when a script should perform an update step and shortly before a script is going to die. Please note that this was only an example of the naming convention used within PixelLight. Within the script system there are no such fixed build in names and you're free to choose your own entry point names. But in general, it makes the life easier to always use the same function names for the same purpose.

**Global Functions added by the C++ Host** Beside writing own global functions directly within the script, the C++ side is able to add global functions during runtime

to expose stuff written in C++ to the script. This allows a script to communicate with its C++ host. In general, global functions added by C++ should be kept to a bare minimum because they introduce script initialization cost and also trash the global script namespace, although there's namespace support to reduce the probability of name conflicts. A far more advanced, universal and powerful way of exposing C++ stuff to the script is by using RTTI objects as discussed in section 2.5. Within the Lua script backend, the RTTI object approach is not adding script initialization cost.

**Functions used as Callbacks** Just calling global function directly is not the only possible way to use them. Global functions can also be used as slots, meaning that when the signal they are connected with is emitted, the global function is called. Please note that this is also true for local functions, local functions can be used as slots as well. See section 2.5.4 for details about slots and section 2.5.5 for information about signals.

As seen within the following source code 2.6, within Lua it's also possible to use a function as a value of a variable. Therefore, a function can also be used as callback,

```

1 function MyFunction()
2   -- ... do some fancy stuff...
3 end
4 myVariable = MyFunction

```

Listing 2.6: Variable with a function as value

name it *slot* if you want. This way, a function pointer can be passed around within your script which can become really handy. Imagine that there's a movie script which accepts such a function pointer as parameter. As soon as the movie playback has been finished this given function is called. The script which is using the movie script can then react on this signal/event and change, for instance, into an interactive mode. Source code 2.7 summes this up.

## 2.4. OOP

First at all, please note that this section has nothing to do with PixelLight RTTI objects which are explained in section 2.5.

Let's directly jump into the topic. The OOP Lua skeleton shown in source code 2.8 has proven to be useful.

```

1  -- [-----]
2  -- [ Includes ]
3  -- [-----]
4  -- Include another Lua script
5  require("FilenameOfAnotherLuaScript")
6
7

```

## 2. Script

```
1 function MyCallbackFunction()
2     -- ... do some fancy stuff...
3 end
4
5 function MyMovie(callbackFunction)
6     -- ... do some fancy stuff...
7
8     -- Movie playback has been finished
9     if callbackFunction ~= nil then
10         callbackFunction()
11     end
12 end
13
14 -- Playback a movie
15 MyMovie(MyCallbackFunction)
```

Listing 2.7: Function as callback

```
8  --[-----]
9  --[ Classes ]
10 --[-----]
11 --@brief
12 -- My Lua class
13 MyClass = {
14
15
16  --[-----]
17  --[ Public definitions ]
18  --[-----]
19  --@brief
20  -- Interaction mode
21  Mode = {
22      WALK = 0, -- Walk mode
23      FREE = 1, -- Free mode
24  },
25
26
27  --@brief
28  -- The default constructor - In Lua a static method
29  new = function()
30
31
32  --[-----]
```

```

33  --[ Private class attributes ]
34  --[-----]
35  -- A private class attribute -> Emulates the C++ "this"-pointer by
using a Lua table
36  local this = {}
37  -- The current interaction mode
38  local _mode = Interaction.Mode.WALK
39
40
41  --[-----]
42  --[ Private class methods ]
43  --[-----]
44  --@brief
45  -- Very secret stuff is happening in this private method
46  local function MyPrivateMethod()
47      -- ... do some fancy stuff...
48  end
49
50
51  --[-----]
52  --[ Public class methods ]
53  --[-----]
54  --@brief
55  -- A public method
56  function this.MyPublicMethod()
57      -- ... do some fancy stuff...
58  end
59
60
61  --[-----]
62  --[ Public class constructor implementation ]
63  --[-----]
64  -- ... do some fancy stuff...
65
66
67  -- Return the created class instance
68  return this
69 end
70
71
72 }

```

Listing 2.8: OOP Lua skeleton

## 2. Script

A new instance of this class can be created by writing `myClassInstance = MyClass.new()` and methods can be called by writing `myClassInstance.MyPublicMethod()`<sup>2</sup>.

Wait a moment, the class definition within source code 2.8 just looks like a namespace as described within section 2.1... and hey, Lua doesn't have OOP support at all!

That's true, Lua has no build in OOP support, but that's no reason to don't use OOP. It's possible to realize OOP by using, yes your guess was right, tables. Even information hiding is possible.

At this point you may possibly ask yourself:

"Why use OOP at all within a script? Shouldn't make a script language the life of a programmer much easier? Why can't we just use variables and functions and be happy?"

The answer is short: No one is forcing you to use OOP. If you're uncomfortable or unfamiliar with OOP and don't want to get into the topic, then don't. If your scripted application gets bigger and you want a powerful tool to write well structured, reusable script code without a totally flooded global namespace due to thousands of global variables and functions, use the mentioned OOP approach. Personally, I wouldn't want to work without OOP because I'am a lazy person.

## 2.5. RTTI Objects

RTTI objects is, were the scripting gets really interesting because it's possible to directly access and manipulate PixelLight RTTI C++ objects by using a script. Within the Lua script backend, RTTI objects are realized by using the *user data* feature of Lua together with Lua's metatables. Whenever Lua is not able to resolve for instance a function call, this request is redirected into the script backend implementation which is then accessing the RTTI system of PixelLight.

**this** PixelLight components using scripts dynamically add a global variable named *this* pointing back to the script calling RTTI object. Of course, the type of the RTTI object depends on the use-case. For example when running a scripted stand-alone application by using *PLViewer*, *this* points to an instance of the *PLEngine::ScriptApplication* RTTI class. Another example would be the script scene node modifier called *PLScriptBindings::SNMScript* where *this* points to an instance of *PLScriptBindings::SNMScript*. All RTTI objects are derived from the base class *PLCore::Object* and therefore share a common feature set. To check the RTTI object type, just write within a Lua script `myRTTIObject:IsInstanceOf("PLEngine::ScriptApplication")` in order to figure out whether the RTTI object you currently dealing with is an instance of *PLEngine::ScriptApplication*. In case you've wondered why it's called *this* and not *self* as it's commonly named in Lua scripts: The answer is, that PixelLight

---

<sup>2</sup>Don't get confused by `."` and `":."`. `":."` is a special Lua shortcut. For instance, instead of writing `myClassInstance.MyPublicMethod(myClassInstance)` one can also write `myClassInstance:MyPublicMethod()` to pass `myClassInstance` as parameter into the method `MyPublicMethod`.

is using a generic script language independent script interface. Components like *PLEngine::ScriptApplication* or *PLScriptBindings::SNMScript* don't use e.g. Lua directly, they're only using the generic script language independent script interface. Not every script language is using *self* to identify *this instance*. PixelLight itself is written in C++, so the name *this* was obviously a better choice than *self* because this way, there are not too many different names describing one and the same concept.

### 2.5.1. Properties

Within PixelLight, RTTI class properties are constant RTTI class information which are defined within C++ as seen in source code 2.9 (see PLCore documentation for details).

```

1 #include <PLCore/Base/Object.h>
2 class MyClass : public PLCore::Object {
3     pl_class(MY_RTTI_EXPORT, MyClass, "", PLCore::Object, "Description of
4         my RTTI class")
5         // Properties
6         pl_properties
7         pl_property("MyFirstProperty", "FirstValue")
8         pl_property("MySecondProperty", "SecondValue")
9         pl_properties_end
10        // Constructors
11        pl_constructor_0(MyConstructor, "Default constructor", "")
12    pl_class_end
13 };

```

Listing 2.9: Defining a new RTTI class with properties (C++)

Within a script language, these properties can be accessed by writing `<property value> = <RTTI object>.<property name>`. For example, if your RTTI object instance is named *myRTTIObject* and you want to get the value of the property with the name *MyFirstProperty* then just write

```
value = myRTTIObject.MyFirstProperty
```

### 2.5.2. Attributes

Within PixelLight, RTTI class attributes are member variables of a class which are defined within C++ as seen in source code 2.10 (see PLCore documentation for details).

```

1 // Class definition of MyClass
2 #include <PLCore/Base/Object.h>
3 class MyClass : public PLCore::Object {
4
5     // RTTI interface
6     pl_class(MY_RTTI_EXPORT, MyClass, "", PLCore::Object, "Description")

```

## 2. Script

```
7 // Attributes
8 pl_attribute(MyInt, int, 10, ReadWrite, DirectValue, "Simple integer", "")
9 pl_attribute(MyFloat, float, 3.1415f, ReadOnly, DirectValue, "PI", "")
10 // Constructors
11 pl_constructor_0(MyConstructor, "Default constructor", "")
12 pl_class_end
13
14 // Default constructor
15 public:
16 MyClass() : MyInt(this), MyFloat(this) {}
17
18 };
19
20 // MyClass RTTI implementation (not done within headers)
21 pl_implement_class(MyClass)
```

Listing 2.10: Defining a new RTTI class with attributes (C++)

Within a script language, these attributes can be accessed by writing `<attribute value> = <RTTI object>.<attribute name>` to read a value and by writing `<RTTI object>.<attribute name> = <attribute value>` to write a value. For example, if your RTTI object instance is named `myRTTIObject` and you want to get the value of the attribute with the name `MyInt` then just write

```
value = myRTTIObject.MyInt
```

### 2.5.3. Methods

Within PixelLight, RTTI class methods are member functions of a class which are defined within C++ as seen in source code 2.11 (see PLCore documentation for details).

```
1 // Class definition of MyClass
2 #include <PLCore/Base/Object.h>
3 class MyClass : public PLCore::Object {
4
5 // RTTI interface
6 pl_class(MY_RTTI_EXPORT, MyClass, "", PLCore::Object, "Description of
  my RTTI class")
7 // Constructors
8 pl_constructor_0(MyConstructor, "Default constructor", "")
9 // Methods
10 pl_method_0(MyMethod, pl_ret_type(void), "My method", "")
11 pl_method_1(MyMethodParameter, pl_ret_type(void), int, "My method
  with given parameter", "")
```

```

12     pl_method_2(MyMethodParameters, bool, int, float, "My method with
13     given parameters and return value", "")
14
15     public:
16     MyClass() {}
17     void MyMethod() {}
18     void MyMethodParameter(int nFirst) {}
19     bool MyMethodParameters(int nFirst, float fSecond) { return true; }
20
21 };
22
23 // MyClass RTTI implementation
24 pl_implement_class(MyClass)

```

Listing 2.11: Definition of RTTI class methods with parameters (C++)

Within a script language, these methods can be accessed by writing `<result> = <RTTI object>:<method name>( <parameters> )`. For example, if your RTTI object instance is named `myRTTIObject` and you want to call the method with the name `MyMethodParameters` then just write

```
result = myRTTIObject:MyMethodParameters(42, 21.21)
```

Please note that the result of such an RTTI object method call can be an RTTI object. You're able to call methods of this returned RTTI object as well... and from RTTI objects returned by this call and so on.

### 2.5.4. Slots

Within PixelLight, RTTI class slots are event handlers within a class which are defined within C++ as seen in source code 2.12 (see PLCore documentation for details).

```

1 // Class definition of MyClass
2 #include <PLCore/Base/Object.h>
3 class MyClass : public PLCore::Object {
4
5     // RTTI interface
6     pl_class(MY_RTTI_EXPORT, MyClass, "", PLCore::Object, "Description of
7     my RTTI class")
8     // Constructors
9     pl_constructor_0(MyConstructor, "Default constructor", "")
10    // Slots
11    pl_slot_0(OnMyEvent, "My slot", "")
12    pl_slot_1(OnMyEventParameter, int, "My slot with given parameter", "

```

## 2. Script

```
12 pl_class_end
13
14 // Public methods
15 public:
16     MyClass() : SlotOnMyEvent(this), SlotOnMyEventParameter(this) {}
17     void OnMyEvent() {}
18     void OnMyEventParameter(int nValue) {}
19
20 };
21
22 // MyClass RTTI implementation
23 pl_implement_class(MyClass)
```

Listing 2.12: Definition of RTTI class slots with parameters (C++)

Within a script language, these slots can be connect to and disconnect from a RTTI class signal. Have a look at the following section 2.5.5 to see how this can be done.

### 2.5.5. Signals

Within PixelLight, RTTI class signals are events within a class which are defined within C++ as seen in source code 2.13 (see PLCore documentation for details).

```
1 // Class definition of MyClass
2 #include <PLCore/Base/Object.h>
3 class MyClass : public PLCore::Object {
4
5     // RTTI interface
6     pl_class(MY_RTII_EXPORT, MyClass, "", PLCore::Object, "Description of
7     my RTTI class")
8     // Constructors
9     pl_constructor_0(MyConstructor, "Default constructor", "")
10    // Signals
11    pl_signal_0(MySignal, "My signal", "")
12    pl_signal_1(MySignalParameter, int, "My signal with given parameter"
13    , "")
14    pl_class_end
15
16    // Public methods
17    public:
18    MyClass() {}
19
20 };
21
22 // MyClass RTTI implementation
```

```
21 pl_implement_class(MyClass)
```

Listing 2.13: Defining a RTTI class with signals (C++)

Within a script language, these signals can be accessed by writing `<RTTI object>:<signal name>( <parameters> )` to emit the signal. For example, if your RTTI object instance is named `myRTTIObject` and you want to emit the signal with the name `MySignalParameter` then just write

```
myRTTIObject:MySignalParameter(42)
```

Within a script, signals can be connected to an RTTI object slot (see section 2.5.4 for details about RTTI slots) or to a global function (section 2.3 for information about global functions) with then is used as a script slot. There are two fixed build in script methods for signals:

- `Connect` is used to connect a slot, which is provided as parameter, to the signal
- `Disconnect` is used to disconnect a slot, which is provided as parameter, from the signal

For example, if your RTTI object instance is named `myRTTIObject` and you want to connect the RTTI class slot with the name `MySlot` of the same RTTI object to a signal with the name `MySignalParameter`, then just write

```
myRTTIObject.MySignalParameter.Connect(myRTTIObject.MySlot)
```

As mentioned before, global as well as local script functions can also be used as slots. If the RTTI class signal is emitted, the global script function will be called. Source code 2.14 shows how this can be used in practice.

```
1 -- Global script function which is used as slot
2 function MySlotFunction()
3   -- ... do some fancy stuff...
4 end
5
6 -- Use the script function "MySlotFunction" as slot and
7 -- connect it with the RTTI signal of our RTTI class instance
8 myRTTIObject.MySignalParameter.Connect(MySlotFunction)
9
10 -- Emit the RTTI signal
11 myRTTIObject:MySignalParameter(42)
```

Listing 2.14: Function as slot

### 2.5.6. Creating a RTTI Class Instance via Script

It's possible to create a RTTI class instance via script. This instance is then managed by the script and will automatically be destroyed when it's no longer referenced. For the purpose of creating a RTTI class instance via script we're providing the *PL.ClassManager.CreateByConstructor()*<sup>3</sup> function which can be used as seen within source code 2.15. This example creates an instance of the *PL.Engine::Camcorder* class,

```
1 cppCamcorder = PL.ClassManager.CreateByConstructor("PL.Engine::Camcorder"  
    , "ParameterConstructor", "Param0=\"\" .. tostring(PL.GetApplication()  
    ) .. "\"")  
2 cppCamcorder:StartCamcorderPlayback("MyAwesomeMovie")
```

Listing 2.15: Creating a RTTI Class Instance via Script

the constructor is provided with a pointer to the C++ application class requested via the script binding function *PL.GetApplication()*. The created RTTI class instance can be used as usual as seen in the presented example. You don't need to care about the fact that this instance was created by the script. By using this approach, it's possible to create more complex or performance critical loose components within C++, and then instancing and using it easily from inside a script.

---

<sup>3</sup>See chapter 4 for more information about available script bindings

## 3. Script Backends

This chapter deals with the different script backends shipping with the PixelLight SDK.

### Null

- PixelLight component name: PLScriptNull
- Used Dynamic-Link Library (DLL)s: PLScriptNull.dll
- The null script backend does nothing

### Lua

- This is the preferred script plugin of the PixelLight team because Lua<sup>1</sup> is really easy to use, lightweight and fast. Also the community is very active.
- Lua 5.1.4 backend
- PixelLight component name: PLScriptLua
- Used DLLs: *PLScriptLua.dll* and *lua51.dll*

Here's a list of some links with useful Lua information:

- There's a really useful *Lua reference card for Lua 5.1*<sup>2</sup> on Thomas Lauer's website at [http://thomaslauer.com/comp/Lua\\_Short\\_Reference](http://thomaslauer.com/comp/Lua_Short_Reference)
- *Object Orientation Closure Approach* can be found at <http://lua-users.org/wiki/ObjectOrientationClosureApproach>

### AngelScript

- Currently not within the official PixelLight SDK
- AngelScript 2.20.2 backend<sup>3</sup>
- PixelLight component name: PLScriptAngelScript
- Used DLLs: *PLScriptAngelScript.dll* and *angelscript.dll*

---

<sup>1</sup>Lua can be downloaded from <http://www.lua.org/>

<sup>2</sup><http://thomaslauer.com/download/luarefv51.pdf> - Lua reference card for Lua 5.1

<sup>3</sup>AngelScript can be downloaded from <http://www.angelcode.com/angelscript/>

### 3. Script Backends

Here's a list of some general notes:

- It looks like that AngelScript (2.20.2) has currently no support for namespaces... so right now, an ugly hack is used: e.g. *PL.Timing.GetTimeDifference()* is written within scripts as *PL\_Timing\_GetTimeDifference()*

#### **V8 (ECMA-262 compliant JavaScript engine)**

- Currently not within the official PixelLight SDK
- V8 JavaScript engine 3.3.1 backend<sup>4</sup>
- PixelLight component name: *PLScriptV8*
- Used DLLs: *PLScriptV8.dll* and *v8.dll*

Here's a list of some links with useful JavaScript information:

- *Object Oriented Programming in JavaScript* by Mike Koss can be found at <http://mckoss.com/jscript/object.htm>
- *Private Members in JavaScript* by Douglas Crockford can be found at <http://javascript.crockford.com/private.html>

#### **Python**

- Currently not within the official PixelLight SDK
- Python 2.7.1 backend<sup>5</sup>
- PixelLight component name: *PLScriptPython*
- Used DLLs: *PLScriptPython.dll* and *python27.dll*

Here's a list of some general notes:

- Currently when using namespaces, an ugly hack is used: e.g. *PL.Timing.GetTimeDifference()* is written within scripts as *PL['Timing']['GetTimeDifference']()*

---

<sup>4</sup>V8 JavaScript engine can be downloaded from <http://code.google.com/p/v8/>

<sup>5</sup>Python can be downloaded from <http://www.python.org/>

## 4. Script Bindings Plugin

This chapter is about script bindings in general and in particular about the loose plugin *PLScriptBindings* which exposes certain parts of PixelLight to script languages.

### 4.1. PL

Exposes general PixelLight features to scripts.

**<RTTI object> PL.GetApplication()** Write *PL.GetApplication()* in order to get an instance of the currently used RTTI application class. This can be a null pointer, but usually it isn't.

### 4.2. PL.System

Exposes *PLCore::System*-features to scripts.

**<bool> PL.System.IsWindows()** Write *PL.System.IsWindows()* in order to figure out whether or not the application is currently running on Microsoft (MS) Windows. Returns *true* if we're currently running on a Windows platform, else *false*.

**<bool> PL.System.IsLinux()** Write *PL.System.IsLinux()* in order to figure out whether or not the application is currently running on Linux. Returns *true* if we're currently running on a Linux platform, else *false*.

**<string> PL.System.GetPlatformArchitecture()** Write *PL.System.GetPlatformArchitecture()* in order to figure out the platform architecture (for instance *x86*, *x64*, *armeabi*, *armeabi-v7a* and so on).

### 4.3. PL.Log

Exposes *PLCore::Log*-features to scripts.

**PL.Log.OutputAlways(<string>)** Write *PL.System.OutputAlways(<string>)* in order to write the given string into the log ('always' log level).

## 4. Script Bindings Plugin

**PL.Log.OutputCritical(<string>)** Write *PL.System.OutputCritical(<string>)* in order to write the given string into the log ('critical' log level).

**PL.Log.OutputError(<string>)** Write *PL.System.OutputError(<string>)* in order to write the given string into the log ('error' log level).

**PL.Log.OutputWarning(<string>)** Write *PL.System.OutputWarning(<string>)* in order to write the given string into the log ('warning' log level).

**PL.Log.OutputInfo(<string>)** Write *PL.System.OutputInfo(<string>)* in order to write the given string into the log ('info' log level).

**PL.Log.OutputDebug(<string>)** Write *PL.System.OutputDebug(<string>)* in order to write the given string into the log ('debug' log level).

## 4.4. PL.System.Console

Exposes *PLCore::Console*-features to scripts.

**PL.System.Console.Print(<string>)** Write *PL.System.Console.Print(<string>)* in order to write the given string into the system console.

## 4.5. PL.Timing

Exposes *PLCore::Timing*-features to scripts.

**<float> PL.Timing.GetTimeDifference()** Write *PL.Timing.GetTimeDifference()* in order to get the past time since last frame (seconds).

**<float> PL.Timing.GetFramesPerSecond()** Write *PL.Timing.GetFramesPerSecond()* in order to get the current Frames Per Second (FPS).

**<bool> PL.Timing.IsPaused()** Write *PL.Timing.IsPaused()* in order to receive *true* when the timing is paused, else *false*. If the timing is paused scene nodes, particles etc. are not updated. The timing will still be updated.

**<float> PL.Timing.Pause(<boolean>)** Write *PL.Timing.Pause(<boolean>)* in order to set pause mode. *true* as first parameter when timing should be pause, else *false*.

**<float> PL.Timing.GetTimeScaleFactor()** Write *PL.Timing.GetTimeScaleFactor()* in order to get the time scale factor. The global time scale factor should only be manipulated for debugging. A factor of  $\leq 0$  is NOT allowed because this may cause problems in certain situations, pause the timer instead by hand! Do NOT make the factor *too* (for example  $> 4$ ) extreme, this may cause problems in certain situations!

**<float> PL.Timing.SetTimeScaleFactor(<float>)** Write *PL.Timing.SetTimeScaleFactor(<float>)* in order to set the time scale factor. Time scale as first parameter (a factor of  $\leq 0$  is NOT allowed!). Returns *true* if all went fine, else *false* (maybe the given factor is  $\leq 0$ ?).

## 4.6. PL.ClassManager

Exposes *PLCore::ClassManager*-features to scripts.

**PL.ClassManager.ScanPlugins(<string>, <boolean>, <boolean>)** Write *PL.ClassManager.ScanPlugins(<string>, <boolean>, <boolean>)* in order to scan a directory for compatible plugins and load them in. Directory to search in as first parameter, boolean value deciding whether or not to take sub-directories into account as second parameter, boolean value deciding whether or not its allowed to perform delayed shared library loading to speed up the program start as third parameter. Returns *true* if all went fine, else *false*.

**PL.ClassManager.Create(<string>, <string>)** Write *PL.ClassManager.Create(<string>, <string>)* in order to create a new RTTI class instance by using the default constructor. Name of the RTTI class to create an instance from as first parameter and optional parameter string for the created instance as second parameter. Returns a pointer to the new RTTI class instance or a null pointer if something went wrong (maybe unknown class). Created instance has initially no references, meaning that a script usually automatically destroys the instance when no longer used.

**PL.ClassManager.CreateByConstructor(<string>, <string>, <string>, <string>)** Write *PL.ClassManager.CreateByConstructor(<string>, <string>, <string>, <string>)* in order to create a new RTTI class instance by using a specified constructor. Name of the RTTI class to create an instance from as first parameter, constructor name as second parameter, constructor parameters as third parameter and optional parameter string for the created instance as fourth parameter. Returns a pointer to the new RTTI class instance or a null pointer if something went wrong (maybe unknown class). Created instance has initially no references, meaning that a script usually automatically destroys the instance when no longer used.



## 5. PLCore RTTI Classes

### 5.1. PLCore::Object Class

#### 5.1.1. Methods

**<bool> IsInstanceOf(<string>)** Write `<RTTI object>:IsInstanceOf(<string>)` in order to check if object is instance of a given class. Class name (with namespace) as first parameter. Returns *true* if the object is an instance of the class or one of it's derived classes, else *false*.

**SetAttribute(<string>, <string>)** Write `<RTTI object>:SetAttribute(<string>, <string>)` in order to set an attribute value. Attribute name as first parameter, attribute value as second parameter.

**SetAttributeDefault(<string>)** Write `<RTTI object>:SetAttributeDefault(<string>)` in order to set an attribute to it's default value. Attribute name as first parameter.

**CallMethod(<string>, <string>)** Write `<RTTI object>:CallMethod(<string>, <string>)` in order to call a method. Method name as first parameter, parameters as string (e.g. "Param0='x' Param1='y' ") as second parameter.

**SetValues(<string>)** Write `<RTTI object>:SetValues(<string>)` in order to set multiple attribute values as a string at once. String containing attributes and values as first parameter (e.g. "Name='Bob' Position='1 2 3' ").

**SetDefaultValues()** Write `<RTTI object>:SetDefaultValues()` in order to set all attributes to default.

**<string> ToString()** Write `<RTTI object>:ToString()` in order to get the object as string. Returns string representation of object.

**FromString(<string>)** Write `<RTTI object>:FromString(<string>)` in order to set the object from string. String representation of object as first parameter.

### 5.1.2. Signals

**SignalDestroyed** Object destroyed signal. When this signal is emitted the object is already in the destruction phase and parts may already be invalid. Best to e.g. only update our object pointer.

## 5.2. PLCore::CoreApplication Class

### 5.2.1. Methods

**<RTTI object> GetApplicationContext()** Write *<RTTI object>:GetApplicationContext()* in order to receive the application context.

**Exit(<integer>)** Write *<RTTI object>:Exit(<integer>)* in order to exit the application. Return code for application as first parameter (usually 0 means no error).

## 5.3. PLCore::FrontendApplication Class

### 5.3.1. Methods

**<RTTI object> GetFrontend()** Write *<RTTI object>:GetFrontend()* in order to get the frontend this application is running in.

## 5.4. PLCore::Frontend Class

### 5.4.1. Methods

**Redraw()** Write *<RTTI object>:Redraw()* in order redraw the frontend.

**Ping()** Write *<RTTI object>:Ping()* in order to give the frontend a chance to process Operating System (OS) messages.

**RedrawAndPing()** Write *<RTTI object>:RedrawAndPing()* in order to redraw the frontend and give the frontend a chance to process OS messages.

**<string> GetTitle()** Write *<RTTI object>:GetTitle()* in order receive the frontend title.

**SetTitle(<string>)** Write *<RTTI object>:SetTitle(<string>)* in order to set the frontend title.

**<integer> GetX()** Write `<RTTI object>:GetX()` in order receive the x position of the frontend (in screen coordinates).

**<integer> GetY()** Write `<RTTI object>:GetY()` in order receive the y position of the frontend (in screen coordinates).

**<integer> GetWidth()** Write `<RTTI object>:GetWidth()` in order receive the width of the frontend.

**<integer> GetHeight()** Write `<RTTI object>:GetHeight()` in order receive the height of the frontend.

**<bool> GetToggleFullscreenMode()** Write `<RTTI object>:GetToggleFullscreenMode()` in order to request whether it's allowed to toggle the fullscreen mode using hotkeys. *true* if it's possible to toggle the fullscreen mode using hotkeys, else *false*.

**SetToggleFullscreenMode(<bool>)** Write `<RTTI object>:SetToggleFullscreenMode(<bool>)` in order set whether it's allowed to toggle the fullscreen mode using hotkeys. *true* as first parameter to allow it, else *false*.

**<bool> GetFullscreenAltTab()** Write `<RTTI object>:GetFullscreenAltTab()` in order to request whether it's allowed to use Alt-Tab if fullscreen mode is used. *true* if it's possible to use Alt-Tab if fullscreen mode is used, else *false*.

**SetFullscreenAltTab(<bool>)** Write `<RTTI object>:SetFullscreenAltTab(<bool>)` in order to set whether it's allowed to use Alt-Tab if fullscreen mode is used. *true* as first parameter to allow it, else *false*.

**<bool> IsFullscreen()** Write `<RTTI object>:IsFullscreen()` in order to request whether or not the frontend is currently fullscreen or not. Returns *true* if the frontend is currently fullscreen, else *false*.

**SetFullscreen(<bool>)** Write `<RTTI object>:SetFullscreen(<bool>)` in order to set whether or not the frontend is currently fullscreen or not. *true* as first parameter if the frontend is currently fullscreen, else *false*.

**<bool> IsMouseOver()** Write `<RTTI object>:IsMouseOver()` in order to request whether or not the mouse cursor is currently over the frontend. Returns *true* if the mouse cursor is currently over the frontend, else *false*.

**<integer> GetMousePositionX()** Write `<RTTI object>:GetMousePositionX()` in order to request the current mouse cursor X position inside the frontend, negative value if the mouse cursor isn't currently over the frontend.

**<integer> GetMousePositionY()** Write `<RTTI object>:GetMousePositionY()` in order to request the current mouse cursor Y position inside the frontend, negative value if the mouse cursor isn't currently over the frontend.

**<bool> IsMouseVisible()** Write `<RTTI object>:IsMouseVisible()` in order to request whether or not the mouse cursor is currently visible. Returns *true* if the mouse cursor is currently visible, else *false*.

**SetMouseVisible(<bool>)** Write `<RTTI object>:SetMouseVisible(<bool>)` in order to set the mouse cursor visibility. *true* as first parameter if the mouse cursor shall be visible.

**SetTrapMouse(<bool>)** Write `<RTTI object>:SetTrapMouse(<bool>)` in order to trap the mouse inside the frontend. *true* as first parameter if the mouse should be trapped inside the frontend, else *false*.

## 5.5. PLCore::ApplicationContext Class

### 5.5.1. Methods

**<string> GetExecutableFilename()** Write `<RTTI object>:GetExecutableFilename()` in order to receive the absolute path of application executable (native path style, e.g. on Windows: "C:\MyApplication\x86\Test.exe").

**<string> GetExecutableDirectory()** Write `<RTTI object>:GetExecutableDirectory()` in order to receive the directory of application executable (native path style, e.g. on Windows: "C:\MyApplication\x86").

**<string> GetAppDirectory()** Write `<RTTI object>:GetAppDirectory()` in order to receive the directory of application (native path style, e.g. on Windows: "C:\MyApplication").

**<string> GetStartupDirectory()** Write `<RTTI object>:GetStartupDirectory()` in order to receive the current directory when the application constructor was called (native path style, e.g. on Windows: "C:\MyApplication\x86").

**<string> GetLogFilename()** Write *<RTTI object>:GetLogFilename()* in order to receive the absolute path to log file, empty if log has not been opened (native path style).

**<string> GetConfigFilename()** Write *<RTTI object>:GetConfigFilename()* in order to receive the absolute path to config file, empty if no config is used (native path style).



## 6. PLInput RTTI Classes

See appendix A for information about the virtual standard controller and appendix B for input examples.

### 6.1. PLInput::Controller Class

#### 6.1.1. Attributes

**Type** Controller type.

**Name** Controller name.

**Description** Controller description.

**Active** State of controller.

#### 6.1.2. Signals

**SignalOnActivate** Controller has been activated or deactivated. *true* as parameter if the controller has been activated, else *false*.

**SignalOnControl** Control event has occurred, control as parameter.

**SignalOnChanged** Controller state has changed.

### 6.2. PLInput::Control Class

#### 6.2.1. Methods

**<RTTI object> GetController()** Write *<RTTI object>:GetController()* in order to receive the controller that owns the control, can be a null pointer.

**<EControlType enum> GetType()** Write *<RTTI object>:GetType()* in order to receive the type of control.

## 6. PLInput RTTI Classes

**<bool> IsInputControl()** Write `<RTTI object>:IsInputControl()` in order to check if control is input or output control. Returns *true* if control is input control, *false* if output.

**<string> GetName()** Write `<RTTI object>:GetName()` in order to receive the control name.

**<string> GetDescription()** Write `<RTTI object>:GetDescription()` in order to receive the control description.

## 6.3. PLInput::Axis Class

### 6.3.1. Methods

**<float> GetValue()** Write `<RTTI object>:GetValue()` in order to receive the axis value.

**SetValue(<float>, <bool>)** Write `<RTTI object>:SetValue(<float>, <bool>)` in order to set the axis value. Current value as first parameter. As second parameter *true* if the current value is relative, else *false* if it's a absolute value.

**<bool> IsValueRelative()** Write `<RTTI object>:IsValueRelative()` in order to check whether or not the value is relative. Returns *true* if the current value is relative, else *false* if it's a absolute value.

## 6.4. PLInput::Button Class

### 6.4.1. Methods

**<char> GetCharacter()** Write `<RTTI object>:GetCharacter()` in order to receive the character associated with the button,  
*0* if none.

**<bool> IsPressed()** Write `<RTTI object>:IsPressed()` in order to check whether or not the button is currently pressed. Returns *true*, if the button is currently pressed, else *false*.

**SetPressed(<bool>)** Write `<RTTI object>:SetPressed(<bool>)` in order to set the button status. *true* as first parameter, if the button is pressed, else *false*.

**<bool> IsHit()** Write *<RTTI object>:IsHit()* in order to check whether or not the button was just hit. Returns *true*, if the button has been hit since the last call of this function, else *false*. This will return the hit-state of the button and also reset it immediatly (so the next call to *IsHit()* will return *false*). If you only want to check, but not reset the hit-state of a button, you should call *CheckHit()*.

**<float> CheckHit()** Write *<RTTI object>:GetValue()* in order to check if the button has been hit. *true* as first parameter, if the button has been hit since the last call of this function, else *false*. This function will not reset the hit-state after being called (see *IsHit()*).

## 6.5. *PLInput::Effect Class*

### 6.5.1. Methods

**<float> GetValue()** Write *<RTTI object>:GetValue()* in order to receive the current value.

**SetValue(<float>)** Write *<RTTI object>:SetValue(<float>)* in order to set the effect value. Current value as first parameter.

## 6.6. *PLInput::LED Class*

### 6.6.1. Methods

**<integer> GetLEDs()** Write *<RTTI object>:GetLEDs()* in order to receive the state of all LEDs as a bitfield.

**SetLEDs(<integer>)** Write *<RTTI object>:SetLEDs(<integer>)* in order to set state of all LEDs as a bitfield. LED states as first parameter.

**<bool> IsOn()** Write *<RTTI object>:IsOn()* in order to receive the LED status. Index of LED (0..31) as first parameter. Returns *true* if the LED is currently on, else *false*.

**SetOn(<integer>)** Write *<RTTI object>:SetOn(<integer>)* in order to set the LED status. Index of LED (0..31) as first parameter. *true* as second parameter, if the LED is on, else *false*.



# 7. PLRenderer RTTI Classes

## 7.1. PLRenderer::RendererApplication Class

### 7.1.1. Methods

**<RTTI object> GetPainter()** Write *<RTTI object>:GetPainter()* in order to get the surface painter of the main window. Returns pointer to surface painter of the main window (can be a null pointer).

**SetPainter()** Write *<RTTI object>:SetPainter(<RTTI object>)* in order to set the surface painter of the main window. Pointer to surface painter of the main window (can be a null pointer) as first parameter.



# 8. PLScene RTTI Classes

See appendix C for scene examples.

## 8.1. PLScene::SceneApplication Class

### 8.1.1. Methods

**<RTTI object> GetRootScene()** Write *<RTTI object>:GetRootScene()* in order to get the root scene container, can be a null pointer.

## 8.2. PLScene::SceneNode Class

### 8.2.1. Attributes

**Flags** Flags.

**DebugFlags** Debug flags.

**Position** Position.

**Rotation** Rotation as Euler angles in degree, [0, 360].

**Scale** Scale.

**MaxDrawDistance** Maximum draw distance of the scene node to the camera, if 0 do always draw, if negative, do always draw this node before other.

**AABBMin** Minimum position of the 'scene node space' axis aligned bounding box.

**AABBMax** Maximum position of the 'scene node space' axis aligned bounding box.

**Name** Optional scene node name. If not defined, a name is chosen automatically.

### 8.2.2. Methods

**<RTTI object> GetContainer()** Write *<RTTI object>:GetContainer()* in order to get the scene container the scene node is in or a null pointer if this is the root node.

**SetContainer(<RTTI object>)** Write *<RTTI object>:SetContainer(<RTTI object>)* in order to set the scene container the scene node is in. Scene container this node is in as first parameter. Returns *true* if all went fine, else *false* (Position, rotation, scale etc. are not manipulated, only the container is changed!).

**<RTTI object> GetRootContainer()** Write *<RTTI object>:GetRootContainer()* in order to get the scene root container (this scene container can be the root scene container), a null pointer on error.

**<RTTI object> GetCommonContainer(<RTTI object>)** Write *<RTTI object>:GetCommonContainer(<RTTI object>)* in order to get the common container of this and another scene node. The other scene node as first parameter. Returns the common container, or a null pointer.

**<RTTI object> GetContainerIndex()** Write *<RTTI object>:GetContainerIndex()* in order to get the index of this scene node within the scene node list of the scene container this scene node is in, <0 on failure (e.g. the scene node is not within a scene container).

**<RTTI object> Clone()** Write *<RTTI object>:Clone()* in order to create a clone of this scene node within the scene container this scene node is in. Returns the created clone of this scene node within the same scene container the original scene node is in, null pointer on error. Scene nodes and scene node modifiers with a set *Automatic*-flag will not be cloned. The debug flags of the created clone are set to 0.

**<RTTI object> CloneAtIndex(<integer>)** Write *<RTTI object>:CloneAtIndex(<integer>)* in order to create a clone of this scene node within the scene container this scene node is in at a certain index inside the scene node list. Index position specifying the location within the scene node list where the scene node should be added as first parameter (<0 for at the end). The created clone of this scene node within the same scene container the original scene node is in, null pointer on error. Scene nodes and scene node modifiers with a set *Automatic*-flag will not be cloned. The debug flags of the created clone are set to 0.

**<RTTI object> GetHierarchy()** Write *<RTTI object>:GetHierarchy()* in order to get the scene hierarchy this scene node is linked into. Returns the scene hierarchy this scene node is linked into, a null pointer on error.

**<string> GetAbsoluteName()** Write `<RTTI object>:GetAbsoluteName()` in order to get the unique absolute name of the scene node (for instance "Root.MyScene.MyNode").

**<bool> IsActive()** Write `<RTTI object>:IsActive()` in order to check whether the scene node is active or not. Returns *true* if the scene node is active, else *false*.

**SetActive(<bool>)** Write `<RTTI object>:SetActive(<bool>)` in order to set whether the scene node is active or not. *true* as first parameter if the scene node should be active, else *false* (sets/unsets the *Inactive*-flag).

**<bool> IsVisible()** Write `<RTTI object>:IsVisible()` in order to check whether the scene node is visible or not. Returns *true* if the scene node is visible, else *false* (invisible/inactive). If the scene node is not active it's automatically invisible but the *Invisible*-flag is not touched. *Visible* doesn't mean *currently on screen*, it just means *can be seen in general*.

**SetVisible(<bool>)** Write `<RTTI object>:SetVisible(<bool>)` in order set whether the scene node is visible or not. *true* as first parameter if the scene node should be visible, else *false* (sets/unsets the *Invisible*-flag). See *IsVisible()*-method for more information.

**<bool> IsFrozen()** Write `<RTTI object>:IsFrozen()` in order to check whether the scene node is frozen or not. Returns *true* if the scene node is frozen, else *false*.

**SetFrozen(<bool>)** Write `<RTTI object>:SetFrozen(<bool>)` in order to set whether the scene node is frozen or not. *true* as first parameter if the scene node should be frozen, else *false* (sets/unsets the *Frozen*-flag).

**<bool> IsContainer()** Write `<RTTI object>:IsContainer()` in order to check whether this scene node is a scene container (*PLScene::SceneContainer*) or not. Returns *true* if this scene node is a scene container, else *false*.

**<bool> IsCell()** Write `<RTTI object>:IsCell()` in order to check whether this scene node is a cell (*PLScene::SCCell*) or not. Returns *true* if this scene node is a cell, else *false*.

**<bool> IsPortal()** Write `<RTTI object>:IsPortal()` in order to check whether this scene node is a portal (*PLScene::SNPortal*) or not. Returns *true* if this scene node is a portal, else *false*.

**<bool> IsCamera()** Write `<RTTI object>:IsCamera()` in order to check whether this scene node is a camera (*PLScene::SNCamera*) or not. Returns *true* if this scene node is a camera, else *false*.

## 8. *PLScene* RTTI Classes

**<bool> IsLight()** Write `<RTTI object>:IsLight()` in order to check whether this scene node is a light (*PLScene::SNLight*) or not. Returns *true* if this scene node is a light, else *false*.

**<bool> IsFog()** Write `<RTTI object>:IsFog()` in order to check whether this scene node is a fog (*PLScene::SNFog*) or not. Returns *true* if this scene node is a fog, else *false*.

**<integer> GetNumOfModifiers(<string>)** Write `<RTTI object>:GetNumOfModifiers(<string>)` in order to get the number of modifiers. Optional modifier class name to return the number of instances from as first parameter (if empty return the total number of modifiers).

**<RTTI object> AddModifier(<string>, <string>)** Write `<RTTI object>:AddModifier(<string>, <string>)` in order to add a modifier. Modifier class name of the modifier to add as first parameter and optional parameter string as second parameter. Returns a pointer to the modifier instance if all went fine, else a null pointer (maybe unknown/incompatible modifier).

**<RTTI object> AddModifierAtIndex(<string>, <string>, <integer>)** Write `<RTTI object>:AddModifierAtIndex(<string>, <string>, <integer>)` in order to add a modifier at a certain index inside the scene node modifier list. Modifier class name of the modifier to add as first parameter and optional parameter string as second parameter, optional index position specifying the location within the scene node modifier list where the scene node modifier should be added as third parameter (<0 for at the end). Returns a pointer to the modifier instance if all went fine, else a null pointer (maybe unknown/incompatible modifier).

**<RTTI object> GetModifier(<string>, <integer>)** Write `<RTTI object>:GetModifier(<string>, <integer>)` in order to get a modifier. Modifier class name of the modifier to return as first parameter, optional modifier index as second parameter (used if class name is empty or if there are multiple instances of this modifier class). Returns the requested modifier, a null pointer on error.

**<bool> RemoveModifierByReference(<RTTI object>)** Write `<RTTI object>:RemoveModifierByReference(<RTTI object>)` in order to remove a modifier by using a given reference to the modifier to remove. Modifier to remove as first parameter. Returns *true* if all went fine, else *false* (maybe invalid modifier). After this method succeeded, the given reference is no longer valid.

**<bool> RemoveModifier(<string>, <integer>)** Write `<RTTI object>:RemoveModifier(<string>, <integer>)` in order to remove a modifier. Modifier class name of the modifier to remove as first parameter, modifier index as second

parameter (used if class name is empty or if there are multiple instances of this modifier class). Returns *true* if all went fine, else *false* (maybe invalid modifier).

**ClearModifiers()** Write *<RTTI object>:ClearModifiers()* in order to clear all modifiers.

**<bool> Delete(<bool>)** Write *<RTTI object>:Delete(<bool>)* in order to delete this scene node. If the first parameter is *true* the scene node will also be deleted when it's protected. Returns *true* when all went fine, else *false*.

**<RTTI object> GetInputController()** Write *<RTTI object>:GetInputController()* in order to get the input controller. Returns the input controller (can be a null pointer).

#### 8.2.3. Signals

**SignalDestroy** Scene node destruction signal. Unlike *PLCore::Object::SignalDestroyed*, the scene node is still intact at the point the signal is emitted.

**SignalActive** Scene node active state change signal.

**SignalVisible** Scene node visible state change signal.

**SignalContainer** Scene node parent container change signal.

**SignalAABoundingBox** Scene node axis aligned bounding box change signal.

**SignalInit** Scene node initialization signal.

**SignalDeInit** Scene node de-initialization change signal.

**SignalAddedToVisibilityTree** Scene node was added to a visibility tree signal. Visibility node representing this scene node within the visibility tree as parameter.

## 8.3. *PLScene::SceneNodeModifier* Class

### 8.3.1. Attributes

**Flags** Flags.

### 8.3.2. Methods

**<RTTI object> GetSceneNode()** Write *<RTTI object>:GetSceneNode()* in order to get the owner scene node.

**<RTTI object> GetSceneNodeIndex()** Write *<RTTI object>:GetSceneNodeIndex()* in order to get the index of this scene node modifier within the scene node modifier list of the owner scene node, <0 on failure.

**<RTTI object> Clone()** Write *<RTTI object>:Clone()* in order to create a clone of this scene node modifier within the owner scene node. Returns the created clone of this scene node modifier within the owner scene node, null pointer on error.

**<RTTI object> CloneAtIndex(<integer>)** Write *<RTTI object>:CloneAtIndex(<integer>)* in order to create a clone of this scene node modifier within the owner scene node at a certain index inside the scene node modifier list. Index position specifying the location within the scene node modifier list where the scene node modifier should be added as first parameter (<0 for at the end). Returns the created clone of this scene node modifier within the owner scene node, null pointer on error.

**<string> GetAbsoluteName()** Write *<RTTI object>:GetAbsoluteName()* in order to get an unique absolute name for the scene node modifier. The name is constructed by using *<absolute owner scene node name>:<scene node modifier class name>.<zero based index>* (for instance *Root.MyScene.MyNode:SNMRotationLinearAnimation.0*). Do not use this method on a regular basis.

**<bool> IsActive()** Write *<RTTI object>:IsActive()* in order to check whether the scene node modifier is active or not. Returns *true* if the scene node modifier is active, else *false*.

**SetActive(<bool>)** Write *<RTTI object>:SetActive(<bool>)* in order to set whether the scene node modifier is active or not. *true* as first parameter if the scene node modifier should be active, else *false* (sets/unsets the *Inactive*-flag).

**<RTTI object> GetInputController()** Write *<RTTI object>:GetInputController()* in order to get the input controller. Returns the input controller (can be a null pointer).

## 8.4. PLScene::SceneContainer Class

### 8.4.1. Attributes

**Hierarchy** Class name of the scene container hierarchy.

**Filename** Filename of the file to load the container from.

## 8.4.2. Methods

**<bool> Clear(<bool>)** Write *<RTTI object>:Clear(<bool>)* in order to destroy all scene nodes within this scene container. If the first parameter is *true* protected scene nodes are destroyed as well. Returns *true* if all went fine, else *false*.

**<RTTI object> GetByIndex(<integer>)** Write *<RTTI object>:GetByIndex(<integer>)* in order to get a scene node by using the given index, result can be a null pointer.

**<RTTI object> GetByName(<string>)** Write *<RTTI object>:GetByName(<string>)* in order to get a scene node by using the given name, result can be a null pointer.

**<RTTI object> Create(<string>, <string>, <string>)** Write *<RTTI object>:Create(<string>, <string>, <string>)* in order to create a new scene node. Name of the scene node class to create an instance from as first parameter, scene node name as second parameter and optional parameter string as third parameter. Returns a pointer to the new scene node or a null pointer if something went wrong (maybe unknown class or the class is not derived from *PLScene::SceneNode*).

**<RTTI object> CreateAtIndex(<string>, <string>, <string>), <integer>** Write *<RTTI object>:CreateAtIndex(<string>, <string>, <string>, <integer>)* in order to create a new scene node at a certain index inside the scene node list. Name of the scene node class to create an instance from as first parameter, scene node name as second parameter and optional parameter string as third parameter, optional index position specifying the location within the scene node list where the scene node should be added as fourth parameter (<0 for at the end). Returns a pointer to the new scene node or a null pointer if something went wrong (maybe unknown class or the class is not derived from *PLScene::SceneNode*).

**CalculateAABoundingBox()** Write *<RTTI object>:CalculateAABoundingBox()* in order to calculate and sets the axis align bounding box in *scene node space*. Because the *scene node space* axis aligned bounding box should always cover all scene nodes of this container, you can use this function to calculate and set this a bounding box automatically.

**LoadByFilename(<string>, <string>, <string>))** Write *<RTTI object>:LoadByFilename()* in order to load a scene from a file given by filename. Scene filename as first parameter, optional load method parameters as second

## 8. *PLScene RTTI Classes*

parameter, optional name of the load method to use as third parameter. Returns *true* if all went fine, else *false*.

### 8.4.3. Signals

**SignalLoadProgress** Scene load progress signal. Current load progress as parameter - if not within 0-1 loading is done.

# 9. PLEngine RTTI Classes

## 9.1. PLEngine::EngineApplication Class

### 9.1.1. Methods

**<RTTI object> GetScene()** Write *<RTTI object>:GetScene()* in order to get the scene container (the *concrete scene*), can be a null pointer.

**SetScene(<RTTI object>)** Write *<RTTI object>:SetScene(<RTTI object>)* in order to set the scene container (the *concrete scene*). New scene container as first parameter (can be a null pointer).

**ClearScene()** Write *<RTTI object>:ClearScene()* in order to clear the scene, after calling this method the scene is empty.

**<bool> LoadScene(<string>)** Write *<RTTI object>:LoadScene(<string>)* in order to load a scene. Filename of the scene to load as first argument. Returns *true* if all went fine, else *false*. This method will completely replace the current scene.

**<RTTI object> GetCamera()** Write *<RTTI object>:GetCamera()* in order to get the scene camera, can be a null pointer.

**SetCamera(<RTTI object>)** Write *<RTTI object>:SetCamera(<RTTI object>)* in order to set the scene camera. New scene camera as first parameter (can be a null pointer).

**<RTTI object> GetInputController()** Write *<RTTI object>:GetInputController()* in order to get the virtual input controller (can be a null pointer). See appendix A for information about the virtual standard controller and appendix B for input examples.

**SetInputController(<RTTI object>)** Write *<RTTI object>:SetInputController(<RTTI object>)* in order to set the virtual input controller. Virtual input controller (can be a null pointer) as first parameter.

**<RTTI object> GetSceneRendererTool()** Write *<RTTI object>:GetSceneRendererTool()* in order to get the scene renderer tool.

**<RTTI object> GetRootScene()** Write *<RTTI object>:GetRootScene()* in order to

**<RTTI object> GetScreenshotTool()** Write *<RTTI object>:GetScreenshotTool()* in order to get the screenshot tool.

### 9.1.2. Signals

**SignalCameraSet** A new camera has been set.

**SignalSceneLoadingFinished** Scene loading has been finished successfully.

## 9.2. PLEngine::ScriptApplication Class

### 9.2.1. Attributes

**OnInitFunction** Name of the optional script function called by C++ when the application should initialize itself. Default is *OnInit*.

**OnUpdateFunction** Name of the optional script function called by C++ when the application should update itself. Default is *OnUpdate*.

**OnDeInitFunction** Name of the optional script function called by C++ when the application should de-initialize itself. Default is *OnDeInit*.

### 9.2.2. Methods

**<string> GetBaseDirectory()** Write *<RTTI object>:GetBaseDirectory()* in order to get the base directory of the application (native path style, e.g. on Windows: 'C:\MyApplication\').

**SetBaseDirectory(<string>)** Write *<RTTI object>:SetBaseDirectory(<string>)* in order to set the base directory of the application (e.g. on Windows: 'C:\MyApplication\'). Base directory as the first parameter.

**<string> GetScript()** Write *<RTTI object>:GetScript()* in order to get the used script instance.

**<string> GetScriptFilename()** Write *<RTTI object>:GetScriptFilename()* in order to get the absolute filename of the used script (native path style, e.g. on Windows: 'C:\MyApplication\Main.lua').

**<string> GetScriptDirectory()** Write *<RTTI object>:GetScriptDirectory()* in order to get the absolute directory the used script is in (native path style, e.g. on Windows: 'C:\MyApplication\' if currently the script 'C:\MyApplication\Main.lua' is used).



# 10. PLGui RTTI Classes

## 10.1. PLGui::Gui Class

### 10.1.1. Methods

**SetMouseVisible(<bool>)** Write *<RTTI object>:SetMouseVisible(<bool>)* in order to set the mouse cursor visibility. *true* as first parameter if the mouse cursor shall be visible.

## 10.2. PLGui::Widget Class

### 10.2.1. Attributes

**ID** Widget ID.

**Name** Widget name.

**Title** Widget title.

**Pos** Widget position.

**Size** Widget size.

**Topmost** Topmost state.

**BackgroundColor** Background color.

### 10.2.2. Methods

**<RTTI object> GetGui()** Write *<RTTI object>:GetGui()* in order to get the owner GUI. Returns pointer to GUI object, can be a null pointer.

**<RTTI object> GetContentWidget()** Write *<RTTI object>:GetContentWidget()* in order to get the content widget, can be a null pointer.

**SetTrapMouse(<bool>)** Write *<RTTI object>:SetTrapMouse(<bool>)* in order to trap the mouse inside the widget. *true* as first parameter if the mouse should be trapped inside the widget, else *false*.

### 10.2.3. Signals

**SignalUserMessage** User message, two custom parameters.

**SignalOnThemeChanged** Theme has been changed.

**SignalUpdateContent** Widget content has been changed.

**SignalUpdateChildWidget** Child widget has been changed, child widget as parameter.

**SignalAddChildWidget** Child widget has been added, child widget as parameter.

**SignalRemoveChildWidget** Child widget has been removed, child widget as parameter.

**SignalClose** Widget shall be closed (ALT+F4 or X-Button pressed).

**SignalCreate** Widget has just been created.

**SignalDestroy** Widget is going to be destroyed.

**SignalShow** Widget gets shown.

**SignalHide** Widget gets hidden.

**SignalEnable** Widget gets enabled.

**SignalDisable** Widget gets disabled.

**SignalGetFocus** Widget gets focus.

**SignalLooseFocus** Widget loses focus.

**SignalActivate** Widget has been activated or deactivated (focus-widget has changed), state as parameter.

**SignalDrawBackground** Widget background gets drawn, graphics object used for painting as parameter.

**SignalDraw** Widget gets drawn, graphics object used for painting as parameter.

**SignalMove** Widget gets moved, new widget position as parameter.

**SignalSize** Widget gets resized, new widget size as parameter.

**SignalWindowState** Window state has changed, new window state as parameter.

**SignalEnterFullscreen** Widget enters fullscreen mode.

**SignalLeaveFullscreen** Widget leaves fullscreen mode.

**SignalPreferredSize** Widget has calculated it's preferred size, preferred size as parameter.

**SignalAdjustContent** Widget content has to be adjusted.

**SignalMouseEnter** Mouse enters widget.

**SignalMouseLeave** Mouse leaves widget.

**SignalMouseOver** Mouse-over widget has changed, state as parameter.

**SignalMouseMove** Mouse moves inside the widget, mouse position within the widget as parameter.

**SignalMouseHover** Mouse hovers over the widget.

**SignalMousePosUpdate** Mouse position inside the widget has changed due to the movement of widget, position as parameter.

**SignalMouseButtonDown** Mouse button is pressed, mouse button and mouse position within the widget as parameters.

**SignalMouseButtonUp** Mouse button is released, mouse button and mouse position within the widget as parameters.

**SignalMouseButtonClick** Mouse button has been clicked, mouse button and mouse position within the widget as parameters.

**SignalMouseButtonDoubleClick** Mouse button has been double-clicked, mouse button and mouse position within the widget as parameters.

**SignalMouseWheel** Mouse wheel moved, mouse wheel movement as parameter.

**SignalKeyDown** Key gets pressed, pressed key and modifier keys pressed as parameters.

**SignalKeyUp** Key gets released, released key and modifier keys pressed as parameters.

**SignalHotkey** Hotkey pressed, hotkey ID as parameter.

**SignalDrop** Data has been dropped onto the widget, data as parameter.

## 10.3. **PLGui::GuiApplication Class**

### 10.3.1. **Methods**

**<RTTI object> GetMainWindow()** Write *<RTTI object>:GetMainWindow()* in order to get the main window. Returns pointer to the main window of the application, a null pointer on error.

**SetMainWindow(<RTTI object>)** Write *<RTTI object>:SetMainWindow(<RTTI object>)* in order to set the main window, pointer to the main window of the application (a null pointer is also valid) as first parameter.

# 11. Contact

[contact@pixellight.org](mailto:contact@pixellight.org)

<http://www.pixellight.org>



# A. Virtual Standard Controller

The *virtual standard controller* is an abstraction of physical devices basing on actions, like *jump*, instead of physical buttons, axis and so on. So, only the most common physical devices mouse and keyboard are directly mapped by the virtual standard controller. The rest of the available devices is mapped onto the offered standard controls within this controller.

In this appendix, the global variable *this* points to the C++ RTTI application class instance invoking the script. By writing

```
this:GetInputController()
```

within a Lua script, one can request the instance of the input controller RTTI class the application is using. Within the PixelLight application framework, this is an instance of the virtual standard controller by default. The information in this appendix assumes that the application, your script is written for, is using this default behaviour instead of implementing an individual input controller.

## Mapped Mouse

Name	Type	Description
MouseX	Axis	X axis (movement data, no absolute data)
MouseY	Axis	Y axis (movement data, no absolute data)
MouseWheel	Axis	Mouse wheel (movement data, no absolute data)
MouseLeft	Button	Left mouse button (mouse button 0)
MouseRight	Button	Right mouse button (mouse button 1)
MouseMiddle	Button	Middle mouse button (mouse button 2)
MouseButton4	Button	Mouse button 4
MouseButton5	Button	Mouse button 5
MouseButton6	Button	Mouse button 6
MouseButton7	Button	Mouse button 7
MouseButton8	Button	Mouse button 8
MouseButton9	Button	Mouse button 9
MouseButton10	Button	Mouse button 10
MouseButton11	Button	Mouse button 11
MouseButton12	Button	Mouse button 12

**Mapped Keyboard**

Name	Type	Description
KeyboardBackspace	Button	Backspace
KeyboardTab	Button	Tabulator
KeyboardClear	Button	Clear (not available everywhere)
KeyboardReturn	Button	Return (often the same as "Enter")
KeyboardShift	Button	Shift
KeyboardControl	Button	Control ("Ctrl")
KeyboardAlt	Button	Alt
KeyboardPause	Button	Pause
KeyboardCapsLock	Button	Caps lock
KeyboardEscape	Button	Escape
KeyboardSpace	Button	Space
KeyboardPageUp	Button	Page up
KeyboardPageDown	Button	Page down
KeyboardEnd	Button	End
KeyboardHome	Button	Home
KeyboardLeft	Button	Left arrow
KeyboardUp	Button	Up arrow
KeyboardRight	Button	Right arrow
KeyboardDown	Button	Down arrow
KeyboardSelect	Button	Select (not available everywhere)
KeyboardExecute	Button	Execute (not available everywhere)
KeyboardPrint	Button	Print screen
KeyboardInsert	Button	Insert
KeyboardDelete	Button	Delete
KeyboardHelp	Button	Help (not available everywhere)
Keyboard0	Button	0
Keyboard1	Button	1
Keyboard2	Button	2
Keyboard3	Button	3
Keyboard4	Button	4
Keyboard5	Button	5
Keyboard6	Button	6
Keyboard7	Button	7
Keyboard8	Button	8
Keyboard9	Button	9
KeyboardA	Button	A
KeyboardB	Button	B
KeyboardC	Button	C
KeyboardD	Button	D
KeyboardE	Button	E
KeyboardF	Button	F

KeyboardG	Button	G
KeyboardH	Button	H
KeyboardI	Button	I
KeyboardJ	Button	J
KeyboardK	Button	K
KeyboardL	Button	L
KeyboardM	Button	M
KeyboardN	Button	N
KeyboardO	Button	O
KeyboardP	Button	P
KeyboardQ	Button	Q
KeyboardR	Button	R
KeyboardS	Button	S
KeyboardT	Button	T
KeyboardU	Button	U
KeyboardV	Button	V
KeyboardW	Button	W
KeyboardX	Button	X
KeyboardY	Button	Y
KeyboardZ	Button	Z
KeyboardNumpad0	Button	Numpad 0
KeyboardNumpad1	Button	Numpad 1
KeyboardNumpad2	Button	Numpad 2
KeyboardNumpad3	Button	Numpad 3
KeyboardNumpad4	Button	Numpad 4
KeyboardNumpad5	Button	Numpad 5
KeyboardNumpad6	Button	Numpad 6
KeyboardNumpad7	Button	Numpad 7
KeyboardNumpad8	Button	Numpad 8
KeyboardNumpad9	Button	Numpad 9
KeyboardNumpadMultiply	Button	Numpad Multiply
KeyboardNumpadAdd	Button	Numpad Add
KeyboardNumpadSeparator	Button	Numpad Separator
KeyboardNumpadSubtract	Button	Numpad Subtract
KeyboardNumpadDecimal	Button	Numpad Decimal
KeyboardNumpadDivide	Button	Numpad Divide
KeyboardF1	Button	F1
KeyboardF2	Button	F2
KeyboardF3	Button	F3
KeyboardF4	Button	F4
KeyboardF5	Button	F5
KeyboardF6	Button	F6
KeyboardF7	Button	F7
KeyboardF8	Button	F8

## A. Virtual Standard Controller

KeyboardF9	Button	F9
KeyboardF10	Button	F10
KeyboardF11	Button	F11
KeyboardF12	Button	F12
KeyboardNumLock	Button	NumLock
KeyboardScrollLock	Button	ScrollLock
KeyboardCircumflex	Button	Circumflex

## Main Character Controls

Name	Type	Description
TransX	Axis	X translation axis: Strafe left/right (+/-)
TransY	Axis	Y translation axis: Move up/down (+/-)
TransZ	Axis	Z translation axis: Move forwards/backwards (+/-)
Pan	Button	Keep pressed to pan
PanX	Axis	X pan translation axis: Strafe left/right (+/-)
PanY	Axis	Y pan translation axis: Move up/down (+/-)
PanZ	Axis	Z pan translation axis: Move forwards/backwards (+/-)
RotX	Axis	X rotation axis: Pitch (also called 'bank') change is moving the nose down and the tail up (or vice-versa)
RotY	Axis	Y rotation axis: Yaw (also called 'heading') change is turning to the left or right
RotZ	Axis	Z rotation axis: Roll (also called 'attitude') change is moving one wingtip up and the other down
Rotate	Button	Keep pressed to rotate
Forward	Button	Move forwards
Backward	Button	Move backwards
Left	Button	Move (rotate) left
Right	Button	Move (rotate) right
StrafeLeft	Button	Strafe left
StrafeRight	Button	Strafe right
Up	Button	Move up
Down	Button	Move down
Run	Button	Keep pressed to run
Crouch	Button	Keep pressed to crouch
Jump	Button	Jump
Zoom	Button	Keep pressed to zoom
ZoomAxis	Axis	Zoom axis to zoom in or out (+/-)
Button1	Button	Button for action 1

Button2	Button	Button for action 2
Button3	Button	Button for action 3
Button4	Button	Button for action 4
Button5	Button	Button for action 5

## Interaction

Name	Type	Description
Pickup	Button	Keep pressed to pickup
Throw	Button	Throw the picked object
IncreaseForce	Button	Keep pressed to increase the force applied to the picked object
DecreaseForce	Button	Keep pressed to decrease the force applied to the picked object
PushPull	Axis	Used to push/pull the picked object



## B. Input Examples

In this appendix, the global variable *this* points to the C++ RTTI application class instance invoking the script

The following Lua script creates a new box when the space key is hit, as soon as the backspace key is hit a box is deleted. When pressing the escape key, the application shuts down. See appendix A for information about the virtual standard controller.

```
1 numOfBoxes = 0
2
3 function OnInit()
4     this:ClearScene()
5     local sceneContainer = this:GetScene()
6     if sceneContainer ~= nil then
7         sceneContainer:Create("PLScene::SNMesh", "Floor", "Position='0 -2.1
8         5' Scale='4 0.1 4' Rotation='0 180 0' Mesh='Default'")
9         sceneContainer:Create("PLScene::SNDirectionalLight", "Sun", "
10        Rotation='45 0 0'")
11        local camera = sceneContainer:Create("PLScene::SNCamera", "
12        FreeCamera")
13        if camera ~= nil then
14            camera:AddModifier("PLEngine::SNMEgoLookController")
15            camera:AddModifier("PLEngine::SNMMoveController")
16        end
17        this:SetCamera(camera)
18    end
19    local inputController = this:GetInputController()
20    if inputController ~= nil then
21        inputController.SignalOnControl.Connect(OnControl)
22    end
23 end
24
25 function OnControl(control)
26     if control:IsInstanceOf("PLInput::Button") and control:IsPressed()
27     then
28         if control:GetName() == "KeyboardEscape" then
29             this:Exit(0)
30         elseif control:GetName() == "KeyboardSpace" then
31             AddBox()
```

## B. Input Examples

```
28     elseif control:GetName() == "KeyboardBackspace" then
29         RemoveBox()
30     end
31 end
32 end
33
34 function AddBox()
35     if numOfBoxes < 10 then
36         numOfBoxes = numOfBoxes + 1
37         local sceneContainer = this:GetScene()
38         if sceneContainer ~= nil then
39             local sceneNode = sceneContainer:Create("PLScene::SNMesh")
40             if sceneNode ~= nil then
41                 sceneNode.Name = "Box_"..numOfBoxes
42                 sceneNode.Scale = "0.5 0.5 0.5"
43                 sceneNode.Mesh = "Default"
44                 sceneNode.Position = "0 -1.5 "..(numOfBoxes+5)
45             end
46         end
47     end
48 end
49
50 function RemoveBox()
51     if numOfBoxes > 0 then
52         local sceneContainer = this:GetScene()
53         if sceneContainer ~= nil then
54             local sceneNode = sceneContainer:GetByName("Box_"..numOfBoxes)
55             if sceneNode ~= nil then
56                 sceneNode:Delete()
57             end
58         end
59         numOfBoxes = numOfBoxes - 1
60     end
61 end
```

# C. Scene Examples

This appendix contains several examples and code snipes regarding to scenes. By using the shown features, one can

- Construct a scene completely by using a script
- Load in a complete scene at once by using a script
- Use a hybrid approach, construct parts by using a script while other sub-scenes are loaded in

In this appendix, the global variable *this* points to the C++ RTTI application class instance invoking the script

## C.1. Constructing Scenes

```
1 function OnInit()  
2   this:ClearScene()  
3   local sceneContainer = this:GetScene()  
4   if sceneContainer ~= nil then  
5     sceneContainer:Create("PLScene::SNMesh", "Floor", "Position='0 -2.1  
6     5' Scale='4 0.1 4' Rotation='0 180 0' Mesh='Default'")  
7     sceneContainer:Create("PLScene::SNMesh", "Box", "Position='0 -1.5 5'  
8     Scale='0.5 0.5 0.5' Mesh='Default'")  
9     sceneContainer:Create("PLScene::SNDirectionalLight", "Sun", "  
10    Rotation='45 0 0'")  
11    local camera = sceneContainer:Create("PLScene::SNCamera", "  
12    FreeCamera")  
13    if camera ~= nil then  
14      camera:AddModifier("PLEngine::SNMEgoLookController")  
15      camera:AddModifier("PLEngine::SNMMoveController")  
16    end  
17    this:SetCamera(camera)  
18  end  
19 end
```

Instead of

## C. Scene Examples

```
1 sceneContainer:Create("PLScene::SNMesh", "Floor", "Position='0 -2.1 5'  
   Scale='4 0.1 4' Rotation='0 180 0' Mesh='Default'")
```

one could also write

```
1 local sceneNode = sceneContainer:Create("PLScene::SNMesh")  
2 if sceneNode ~= nil then  
3   sceneNode.Name = "Floor"  
4   sceneNode.Position = "0 -2.1 5"  
5   sceneNode.Scale = "4 0.1 4"  
6   sceneNode.Mesh = "Default"  
7 end
```

... but the first version is more handy in this situation.

Have a look at *45ScriptApplication.lua* within the PixelLight SDK for a more advanced example on how to construct complete scenes by using a script.

## C.2. Loading Scenes

In this section we're going to use the plugin *PLAssimp* using Open Asset Import Library (ASSIMP) to be able to load in a complete scene, without merging everything into one huge mesh. The following example is using the x-scene from <http://www.daniel-sadowski.com/nyx/>.

Download [http://www.daniel-sadowski.com/nyx/nyx\\_exec.exe](http://www.daniel-sadowski.com/nyx/nyx_exec.exe) and extract it. Then create a Lua script file within the same directory as "Nyx.exe" with the following content:

```
1 function OnInit()  
2   this:SetBaseDirectory("data/tex/") -- The loaded scene is using just  
   texture names like "Houses"  
3   this:ClearScene()  
4   local sceneContainer = this:GetScene()  
5   if sceneContainer ~= nil then  
6     sceneContainer:Create("PLScene::SceneContainer", "Serenael", "Scale  
   ='0.05 0.05 0.05' Filename='data/msh/serenael.x'")  
7     sceneContainer:Create("PLScene::SNDirectionalLight", "Sun", "  
   Rotation='45 0 0'")  
8     local camera = sceneContainer:Create("PLScene::SNCamera", "  
   FreeCamera")  
9     if camera ~= nil then  
10      camera:AddModifier("PLEngine::SNMEgoLookController")  
11      camera:AddModifier("PLEngine::SNMMoveController")  
12    end  
13    this:SetCamera(camera)  
14  end
```

```
15 end
```

... and just load this script with PLViewer and you should be able to fly through this fantasy city within the PixelLight viewer.

You can also replace

```
1 sceneContainer:Create("PLScene::SceneContainer", "Serenael", "Scale
   ='0.05 0.05 0.05' Filename='data/msh/serenael.x'")
```

by

```
1 sceneContainer:Create("PLScene::SNMesh", "Serenael", "Scale='0.05 0.05
   0.05' Mesh='data/msh/serenael.x'")
```

in order to load in this x-scene as one single huge mesh. By the way, ASSIMP was able to detect 80 mesh instances in this scene, meaning it optimizes the data quite well.

*PLScene::SceneContainer* offers a RTTI method named *LoadByFilename()*. With this method, one can now also write within a Lua script e.g.

```
1 local serenaelContainer = sceneContainer:Create("PLScene::SceneContainer
   ", "Serenael", "Scale='0.05 0.05 0.05'")
2 if serenaelContainer ~= nil then
3   serenaelContainer:LoadByFilename("data/msh/serenael.x", "Quality=1")
4 end
```

in order to load in a scene and proving the loader with additional parameters, or even decide which concrete loader method should be used by writing

```
1 local serenaelContainer = sceneContainer:Create("PLScene::SceneContainer
   ", "Serenael", "Scale='0.05 0.05 0.05'")
2 if serenaelContainer ~= nil then
3   serenaelContainer:LoadByFilename("data/msh/serenael.x", "Quality=1", "
   LoadParams")
4 end
```

Due to the flexibility the RTTI introduces, the available loader parameters depend on the used loader implementation. If there's no need for additional settings,

```
1 local serenaelContainer = sceneContainer:Create("PLScene::SceneContainer
   ", "Serenael", "Scale='0.05 0.05 0.05'")
2 if serenaelContainer ~= nil then
3   serenaelContainer:LoadByFilename("data/msh/serenael.x")
4 end
```

or

```
1 sceneContainer:Create("PLScene::SceneContainer", "Serenael", "Scale
   ='0.05 0.05 0.05' Filename='data/msh/serenael.x'")
```

... also does the job.



# Abbreviations

**DLL** Dynamic-Link Library, under Linux it's called Shared Object (SO)

**SO** Shared Object, under MS Windows it's called DLL

**API** Application Programming Interface

**SDK** Software Development Kit, also known as *devkit*

**OS** Operating System

**GUI** Graphical User Interface

**RTTI** Run-Time Type Information, or Run-Time Type Identification

**ASSIMP** Open Asset Import Library

**OOP** Object-Oriented Programming

**MS** Microsoft

**FPS** Frames Per Second