Contents

1 Introduction **1**

1.1 Guide to the Code 1
1.2 Guide to the manual 1
1.3 Reader's Guide to the Manual 1

2 Scene Description Language **3**

2.1 RubySceneGraph language 3
2.2 File structure 4
2.3 Node Expression 4
2.4 Scene Graph templates 5
2.5 Language Reference 6

3 Development Conventions **7**

3.1 Naming conventions 7
3.2 CVS Tagging 7

4 Zeitgeist application framework **9**

4.1 Writing a class object for a C++ class 9
4.2 Registering a Class Object 11
4.2.1 Direct Registration 11
4.2.2 Indirect Registration 12
4.3 Exposing C++ functions to Ruby Scripts 12

Introduction 1

1.1 Guide to the Code

- Salt
- Zeitgeist
- Oxygen
- Kerosin

1.2 Guide to the manual

1.3 Reader's Guide to the Manual

Scene Description Language 2

SimSpark provides access to the managed scene graph in several ways. Besides the internal C++ interface and external access via Ruby script language, an extensible mechanism for scene description languages is implemented. This allows for both a procedural and a description-based scene setup.

A scene is imported using one of any number of registered scene importer plugins, each supporting a different scene description language.

2.1 RubySceneGraph language

Currently one S-expression-based importer is implemented. This language is called *RubySceneGraph* (RSG for short) and used to model the current robot models. It maps the scene graph structure to the nesting of Lisp-like `s-expressions`.

An s-expression is a list of elements. Each element is either an `atom` or is itself another `list` of atoms. An atom is either a predefined keyword or a non empty string literal that has no further syntactic structure. The syntax of s-expressions, written using EBNF is given in Listing 2.1.

```
character    -> "A" | ... | "Z" | "1" | ... | "9",
atom         -> character+
list         -> "(" s_expression* ")"
s_expression -> atom | list
```

Listing 2.1: *EBNF notation of s-expressions*

On the semantic side the *RubySceneGraph* interpreter recognizes a set of special atoms. The first atom in each subexpression determines its type. The set of keywords comprises some atoms that allow the interpreter to distinguish different expression types.

For most expression types exists a short hand notation that can be used to save some typing. The short hand notation is further used in the monitor protocol to keep it more compact.

- The `RubySceneGraph` expression is the header expression of every scene graph file.
- The `node` (short `nd`) expression declares a new scene graph node.
- The `importScene` expression is replaced with the content of another scene graph file.
- The `template` (short `templ`) expression declares a set of parameters for a following scene fragment that can later be reused like a macro.

- The `define` (short `def`) expression defines a variable with the scope of the current scene file and all files sourced with the `importScene` expression.
- The `eval` expression uses the ruby interpreter to evaluate an expresion and is replaced by the computed value.
- Every other expression type is interpreted as a `method call` that is carried out by the ruby Script interface

Apart from the different expression types listed above a replacement mechanism is implemented. Every atom literal starting with a dollar sign is interpreted as a template or variable parameter and replaced with its actual value.

We shall describe the semantic of the different expression types below together with some small usage examples and a partial reference of available node types and methods.

2.2 File structure

The top level structure of a ruby scene file consists of two s-expressions. The first expression must be the header expression. It allows the parser to confirm the file type and to get information about the version of the used language.

The syntax of the header expression is `(RubySceneGraph <major Version> <minor Version>)`. Currently the only valid header states `0` for the major and `1` for the minor version.

The header is followed by a single s-expression that contains the scene graph body. Any further expression is discarded. The body expression consists of an optional single template expression and a set of node expressions. The resulting structure is outlined in listing 2.2. Note that lines starting with a semicolon are comment lines.

```
; the header expression
(RubySceneGraph 0 1)
(
 ; the body of the file starts here

 ; declare this file as a template
 (template \$lenX \$lenY \$lenZ \$density \$material)

 ; compute the volume of the box
 (define \$volume eval (\$lenX * \$lenY * \$lenZ)

 ; declare the top level scene graph node
 (node Box
       ; children of the top level node go here
       (node DragController
       )
 )
)
```

Listing 2.2: *File Structure*

2.3 Node Expression

The scene graph consists of a tree of object instances, called nodes. Each node in the scene graph is declared with the `(node <ClassName>)` expression. The *ClassName* argument gives the name of a class registered to the Zeitgeist class factory system.

The semantic of a node expression is to instantiate a new scene graph object of the given class type. The importer therefore relies on the Zeitgeist class factory system to create the requested object. It is then installed as a child of the nearest enclosing node expression. If there is no enclosing node expression then the node is a top level node of the expressed scene graph.

The set of top level nodes are installed as children of the node below which the current graph is imported. This is either the global root node of the system, or an insertion point defined with the importScene expression within another scene graph file. The nesting of node expressions therefore defines directly the structure of the resulting scene graph with a very small syntactic overhead.

The set of top level nodes are installed as children of the node below which the current graph is imported. This is either the global root node of the system, or an insertion point defined with the importScene expression within another scene graph file. The nesting of node expressions therefore defines directly the structure of the resulting scene graph with a very small syntactic overhead.

2.4 Scene Graph templates

The language further allows the reuse of scene graph parts in a macro like fashion. This enables the construction of a repository of predefined partial scenes, or complete agent descriptions. The macro concept is available through the `(importScene <filename> <parameter>*)` expression. This expression recursively calls the importer facilities of the system. It takes the nearest enclosing node expression as the relative root node to install the scene graph described within the given file.

Note that the given file must not necessarily be another RubySceneGraph file but any file type registered to the importer framework. This allows the nesting of scene graph parts expressed in different graph description languages. An example application of this feature is that parts of the resulting scene could be created by application programs better suited to create 3D models. By now, we do not exploit this feature yet.

The list of parameters given to the importScene expression is passed on to the responsible importer plugin. If another ruby scene graph file is imported that declares a template, they are substituted with its formal parameters.

A template declaration within the imported file has to meet the following syntax: `(template <parameterName>*)`. A parameter name is a string literal that is prefixed with a dollar sign, see listing 2.2 for an example declaration. All parameter names that follow within the body of the file are replaced with their actual content.

The usage example in listing 2.3 below assumes a `box.rsg` file. It uses that to construct boxes with varying sizes and colors.

```
(RubySceneGraph 0 1)
(
 (node Transform
       (importScene box.rsg 1 3 0.8 10 matRed)
 )
 (node Transform
       (importScene box.rsg 2 4 0.4 8 matBlue)
 )
)
```

Listing 2.3: *importScene example*

2.5 Language Reference

Development Conventions 3

We agreed on a set of conventions that you should apply to when you contribute to the development.

3.1 Naming conventions

This section lists some naming conventions used throughout the source code.

- Class Object Files

  The implementation of zeitgeist class object files in contained in a separate C++ file with a `_c` prefix, see section 4.1. A class called `simple` has it's class object implementation in the file `simple_c.cpp`

- Bundle Registration

  The C++ implementation file that registers classes contained in a zeitgeist bundle to the calling core is called `export.cpp`, see section 4.2

- Function Names

  Ruby functions start with a lower letter, C++ functions with a capital letter

- Class Names

  Common well known service class instances are called servers and are installed below the `/sys/server/` path in the zeitgeist hierarchy. The script service implementation for example is called `ScriptServer` and is installed ad `/sys/server/script`.

3.2 CVS Tagging

Developing and testing an new feature of Simspark is usually done on a separate CVS branch. We implemented the following convention to tag the branch and trunk before and after merging and branching takes place:

Before branching off the trunk is tagged `BRANCHNAME_base` in order to mark the last CVS state before the branch is started.

Each time before a branch is merged back to the trunk the trunk ist tagged `BRANCHNAME_premergeN`. The value `N` increases with each separate merge. After the branch is merged the trunk is tagged `BRANCHNAME_postmergeN` and the merged branched is tagged `BRANCHNAME_mergedN`

Zeitgeist application framework 4

The Zeitgeist library provides two major features. It implements a mechanism to work with class objects in C++. A class object is just a factory of class instances. In addition to this mechanism, it also implements an object hierarchy. This hierarchy is essentially a virtual file system, where the 'directories' and 'files' are instances of C++ classes. These two concepts are quite intertwined with each other, as class objects can also live inside the object hierarchy. Objects within the hierarchy are identified with a unique name. On top of these two features, the Zeitgeist library also provides three very important 'built-in' services. Each basic service is usually represented by a 'server' class in our terminology. An instance of such a class provides the service to other parts of the system. The three services built into the Zeitgeist library are the LogServer, the FileServer, and the ScriptServer.

4.1 Writing a class object for a C++ class

This topic is fundamental in the understanding of how the class object and the object hierarchy framework interact with each other. Let's say we have a simple class:

```
class Simple
{
public:
  Simple();
  virtual ~Simple();

  void DoSomething();
  void PrintString(const std::string& s);
  void PrintInt(int i);
  void PrintFloat(float f);
  void PrintBool(bool b);
};
```

Now, in order to write a class object for this class we must do two things: First, the class must derive from the `zeitgeist::Object` class or one of its descendants, especially `Leaf` and `Node` if instances of this class are to live in the object hierarchy. In addition to this, a class object must be declared and defined, which serves as a factory for instances of this class.

The first step is performed easily:

```
#include <zeitgeist/leaf.h>

class Simple : public zeitgeist::Leaf
{
public:
  Simple();
  virtual ~Simple();

  void DoSomething();
  void PrintString(const std::string& s);
  void PrintInt(int i);
  void PrintFloat(float f);
  void PrintBool(bool b);
};
```

Now, we just have to write a class object for this class. As this is a pretty repetitive procedure, several helper-macros exist to make this as painless as possible. First, we declare the class object. This is done in the header file with the `DECLARE_CLASS()`-macro:

```
#include <zeitgeist/leaf.h>

class Simple : public zeitgeist::Leaf
{
public:
  Simple();
  virtual ~Simple();

  void DoSomething();
  void PrintString(const std::string& s);
  void PrintInt(int i);
  void PrintFloat(float f);
  void PrintBool(bool b);
};

DECLARE_CLASS(Simple);
```

With this macro, we declare the class object. If `Simple` would have been an abstract base class (containing one or more pure virtual functions) we would have needed to use the `DECLARE_ABSTRACTCLASS()`-macro instead.

Both of these macros create a new class with the mangled name `Class_XXXX`, where `XXXX` is the name of the class. In our case this would be `Class_Simple`. This class is derived from `zeitgeist::Class`. In the case of `DECLARE_CLASS()` the macro also provides a `CreateInstance()` function, which creates an instance of the Simple class.

The `DECLARE_ABSTRACTCLASS()` macro does not do this, as it is impossible to create an instance of an abstract class. It inherits the base behavior from zeitgeist::Class, which just returns `NULL`.

In addition to this, both macros declare a `DefineClass()` member function, which needs to be implemented to define the class object fully. This is done in an additional `CPP` file. If the class above was implemented in the files `simple.h` and `simple.cpp`, the accompanying class object should be placed in the file `simple_c.cpp`. This naming convention has been found useful during development and should be adopted. A minimal `simple_c.cpp` should look like this:

```
#include "simple.h"
```

```
using namespace zeitgeist;

void CLASS(Simple)::DefineClass()
{
  DEFINE_BASECLASS(zeitgeist/Leaf);
}
```

The `CLASS()`-macro is used to identify the name of the class object. In the above example, it just resolves to `Class_Simple`. The `DEFINE_BASECLASS()` macro is used to identify the base class of the class described by this class object. This can appear multiple times to allow for multiple inheritance. We now have a working class object. In order to use it it must be registered to the zeitgeist framework. This process is described in section 4.2.

4.2 Registering a Class Object

In section 4.1 we created a class object for the C++ class `Simple`. Before this class object can be used by the zeitgeist framework it must be registered with a `zeitgeist::Core` object.

The Core is the object hierarchy, which is basically a virtual file system where instances of classes represent the directories and 'files'. Therefore, each class instance can be identified by a path. The Core has a function called `RegisterClassObject()` which inserts the class object into the object hierarchy. Class objects are located under the `'/classes/'` branch of the hierarchy. We have two major scenarios of how a class object can be registered: Directly or indirectly!

4.2.1 Direct Registration

This scenario is used in the case of static libraries and executables, which want to expose custom classes to the object hierarchy. An example of a library, which does this is Kerosin. This way of registering is pretty straight forward and involves an initialization function which has a means to access the `zeitgeist::Core` instance where the class objects are to be registered. Then it (directly) calls the `RegisterClassObject()`-method to successively add class objects. Here's a short snippet from the Kerosin library:

```
#include "kerosin.h"
#include <zeitgeist/scriptserver/scriptserver.h>

using namespace kerosin;
using namespace zeitgeist;

Kerosin::Kerosin(zeitgeist::Zeitgeist &zg)
{
  zg.GetCore()->RegisterClassObject(new CLASS(SoundServer), "kerosin/");

  zg.GetCore()->RegisterClassObject(new CLASS(InputServer), "kerosin/");

  zg.GetCore()->RegisterClassObject(new CLASS(ImageServer), "kerosin/");

  zg.GetCore()->RegisterClassObject(new CLASS(FontServer), "kerosin/");

  zg.GetCore()->RegisterClassObject(new CLASS(OpenGLServer), "kerosin/");
```

The Kerosin library is initialized by creating an instance of the Kerosin class. The constructor needs a reference to the Zeitgeist object (which represents the Zeitgeist library), so it can get access to the Core. The class objects are all added to the object hierarchy under the `'/classes/'` branch.

The second parameter of `RegisterClassObject()` allows to specify an additional sub path. So, the class object for the SoundServer (for example) will be located at `'/classes/kerosin/SoundServer'`. In in a way, this allows to create a form of namespaces among the class objects. As we can see, it is possible to add a class object directly at any time during the execution of the program.

4.2.2 Indirect Registration

The second scenario involves packaging a bunch of class objects into a shared library. On Unix platform these files are called `.so` files. In the Windows world thees files are `.DLLs`. The shared library is used as a class library, containing a collection of classes to be added to the object hierarchy. This kind of library is referred to as a `Bundle`.

The Bundle contains a well defined entry point function, which registers its contents with the calling Core. This is simplified through the use of several macros. First, we have to create a shared library project, containing a bunch of classes and their corresponding class objects. This process is described in section 4.1. Then, we create an additional `.CPP` file. By convention this file is called `export.cpp`. Let's say we have two classes in our bundle, `Simple` and `Complex`. The corresponding `export.cpp` would look like this:

```
#include "simple.h"
#include "complex.h"
#include <zeitgeist/zeitgeist.h>

ZEITGEIST_EXPORT_BEGIN()
  ZEITGEIST_EXPORT(Simple);
  ZEITGEIST_EXPORT(Complex);
ZEITGEIST_EXPORT_END()
```

Thanks to the macros, this is again a quite compact notation. We just include the header files of all classes we want to expose. These should also incorporate the correct class object declarations. Then we include the zeitgeist framework. The following macros implement the entry point function. `ZEITGEIST_EXPORT_BEGIN()` implements the beginning of the function. Then we use `ZEITGEIST_EXPORT()` for every class we want to export, and finally `ZEITGEIST_EXPORT_END()` to 'terminate' the function. The compiled library is then a Zeitgeist-capable Bundle. The `ZEITGEIST_EXPORT_EX()` macro can be used to specify a subpath as above with `RegisterClassObject()`.

To indirectly register the classes contained in a Bundle, you just import the bundle into the Core. This is done with the `ImportBundle()`-member function. This function opens the shared library. Gets the entry point function and calls it with an STL list. The class objects are added to this list within the entry point function. After returning, all class objects contained in this list are added to the object hierarchy.

4.3 Exposing C++ functions to Ruby Scripts

As we want to expose much functionality to the script side, we also want to be able to call C++ functions from Ruby. In order to do this, we intercept unknown function calls on the Ruby side.

The parameters are converted on the C++ side. There we know the name of the function to call, the object to call them on, and the parameters.

But, how do we actually call the correct C++ function. The answer lies in the class object. The class object will contain the necessary meta-data to reroute the function call to the correct C++ function. The class object defines the interface of the class to the script side. This is done in the same file as the class definition was performed, i.e. the additional CPP file as described in section. 4.1. Let's go back to the `Simple` class from earlier:

```
#include <zeitgeist/leaf.h>

class Simple : public zeitgeist::Leaf
{
public:
  Simple();
  virtual ~Simple();

  void DoSomething();
  void PrintString(const std::string& s);
  void PrintInt(int i);
  void PrintFloat(float f);
  void PrintBool(bool b);
};

DECLARE_CLASS(Simple);
```

In order to to expose the `DoSomething()` method the `simple_c.cpp` would look like this:

```
#include "simple.h"

using namespace zeitgeist;

FUNCTION(doSomething)
{
  if (in.size() == 0)
  {
    Simple *simple = static_cast<Simple*>(obj);
    simple->DoSomething();
  }
}

void CLASS(Simple)::DefineClass()
{
  DEFINE_BASECLASS(zeitgeist/Leaf);
  DEFINE_FUNCTION(doSomething);
}
```

Every function is declared using the `FUNCTION()`-macro. As a parameter it takes the name of the function. By convention Ruby-side functions start with a lower-case letter and C++-side functions with a capital letter.

The function macro just declares a function, which takes two parameters: `obj` and `in`. `obj` is the object we are calling the function on, i.e. basically, the `this` or `self` pointer. `in` a reference to a `ParameterList` instance. Basically the `Parameterlist` class manages a list of `boost::any` values and provides helper to iterate the list and cast values to the desired type.

In `DefineClass()` we also have to define the function using the `DEFINE_FUNCTION()` macro. After this has been done and after the class object is registered with the Core, we can execute the following script-code:

```
mySimpleObj = new ('Simple', 'test')
mySimpleObj.doSomething
```

This ruby code would then call the C++ member function `DoSomething()`. Now, how about passing some parameters. Let's marshall the PrintInt() function:

```
#include "simple.h"

using namespace zeitgeist;
using namespace boost;

FUNCTION(printInt)
{
  if (in.GetSize() != 1)
  {
      return false;
  }

  int value;
  in.GetValue(in[0]);

  obj->PrintInt(value);
  return true;
}
```

The above function obviously also would need to be defined in `DefineClass()`. Now we see that the first given value is automatically cast to `int` using the `GetValue()` function on the `ParameterList`. The `obj` contains the reference to our `Simple` class instance. Using this reference we call the C++ `PrintInt()` function.

The rest of the code is concerned with error checking as `GetSize()` is used to check the number of the given parameters. In the current implementation every C++ function that is exposed to ruby needs to return a value, so we just return a boolean value here. The return type is not fixed but marshalled using a `boost::any` container. It is therefore possible to return different types.