Computer Graphics Technical Reports

CG-2004/1

MotionLab

User's Guide and Programmer's Guide

Christoph Brzozowski
Stanley Klemme
Joachim Melzer
Hoang Nguyen
Universität Bonn. {brzozows,klemme,melzer,nguyen}@cs.uni-bonn.de

Institut für Informatik II
Universität Bonn
D-53117 Bonn, Germany

ISSN 1610-8892

MotionLab

- User's Guide -

Christoph Brzozowski
Stanley Klemme
Joachim Melzer
Hoang Nguyen

Contents

1 Overview

The *MotionLab* application was designed to serve as a framework for motion editing and analysis. This document gives a short overview about how to use the application. It explains it's user interface and all console commands which are available in the current version of the application.

2 The user interface

After launching the *MotionLab* application the main window appears:

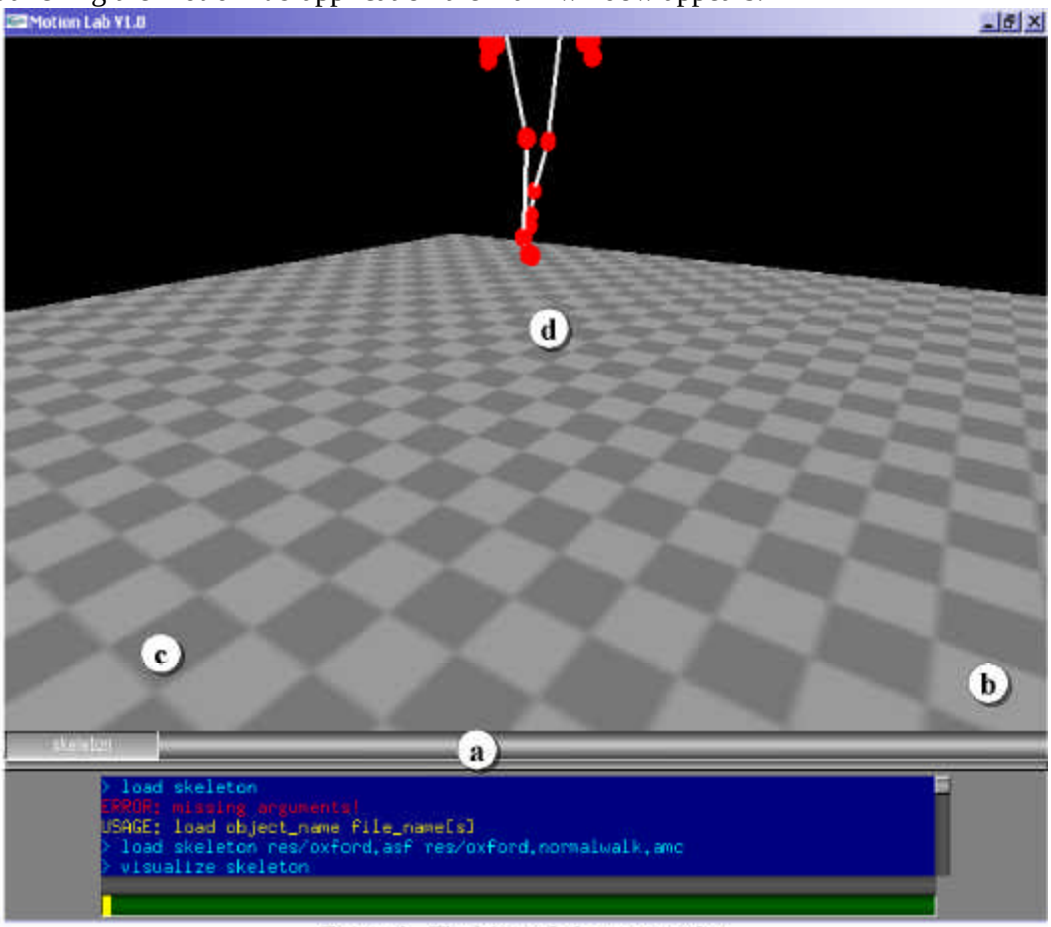

Figure 1 - The MotionLab user interface

The main window is split up into four areas:

- **Console (a)**: The application's console is located at the bottom of the window. Here the user can input commands to control the application's behavior
- **Separator (b)**: The thin line above the console is a separator, which can be moved using the mouse to change the size of the console and the window panel.
- **Window bar (c)**: The window bar contains window buttons. Each button stands for a window which is open at the moment inside the application. The buttons can be used to switch between the windows by clicking with the left mouse button on them. Clicking a button using the right button closes the window and it's window button disappears.
- **Window panel (d)**: The top area of the main window is reserved for application sub windows. Typically this area is used to visualize some motion capture data.

3 The console

The *MotionLab* application can be controlled by calling commands using the console placed at the bottom of the main window. This section will give an overview about all commands which are available in the current version of this application. Some commands expect one or more parameters. Calling those commands without any parameter outputs a text to the console describing which parameters are expected by this command.

Command	*Description*
`exit`	The ***exit*** command terminates the application immediately.
cls	The ***cls*** command clears the contents of the console.
about	The ***about*** command writes information about the application and it's authors to the console.
dump	The ***dump*** command can be used to dump the contents of a specific object contained in the application's object repository. The command name must be entered followed by the name of an object, so that the syntax is: *`dump`* **object_name**
list	The ***list*** command can be used to output a list of objects of a certain type to the console. Which object's can be listed can be obtained by calling *list* without any parameter. This command is described below in detail. It's syntax is: *`list`* **object_name**
calculate_mass	The ***calculate_mass*** command can be used to calculate the mass properties for a skeleton. The command is described below in detail. It's syntax is: *`calculate_mass`* **skeleton_object script_file [algorithm] [parameters]**
describe_algorithm	The ***describe_algorithm*** command can be used to obtain information about a certain mass calculation algorithm. The syntax for this command is: *`describe_algorithm`* **algorithm_name**
load	The ***load*** command can be used to load files from disk into the object repository. Only some specific file types are supported. The syntax for this command is: *`describe_algorithm`* **destination_object [files]**
visualize	The ***visualize*** command can be used to visualize motion capture data. The syntax for this command is: *`visualize`* **[skeleton_objects]**

Command	Description
dir	The ***dir*** command outputs the contents of the current directory to the console. Wildcards are allowed. The syntax for this command is: `dir` **[file_path]**
cd	The ***cd*** command can be used to change the current directory of the application. The syntax for this command is: `cd` **[file_path]**
delete_object	The ***delete_object*** command can be used to delete one or more objects from the application's object repository. The syntax for this command is: `delete_object` **[object_names]**

Table 1 - Available console commands

The contents of the console can be scrolled using the scrollbars at the right and the bottom border of the console. The console has also a command history, so that commands which have been entered previously can be browsed by pressing the up and down cursor keys.

NOTE: Since GLUT is used for the implementation of the user interface, keyboard events are only sent to the window containing the mouse cursor. So if the mouse cursor is outside the console no commands can be entered.

3.1 Listing available console commands

To get a list of all available console commands simply type

```
list commands
```

at the console prompt. A list of all commands and the modules which implement them will be written to the console then.

4 The object repository

The different modules of which the *MotionLab* application consists can exchange data using a central instance called the ***object repository***. The object repository behaves like a simplefile system with only one root directory. Modules can add objects to the repository or remove them from it.

4.1 Listing the contents of the object repository

The contents of the object repository can be listed using the ***list*** command. Simply type

```
list objects
```

at the console prompt and hit the enter/return key. A list of all objects contained in the object repository and their type is written to the console then.

4.2 Deleting objects from the object repository

To delete an object from the object repository the ***delete_object*** command can be used. For

example if the object repository contains an object called ***skeleton1***, so typing

```
delete_object skeleton1
```

at the console prompt will delete it from the object repository. If the object is in use by some module an error message will be issued and the object will not be deleted.

It is also possible to delete more than one object by specifying more than one object name.

5 Working with motion capture data

The *MotionLab* application was especially designed for the analysis and editing of motion capture data. This section will give an overview how motion capture data can be loaded into the application and visualized.

5.1 Loading motion capture data

Motion capture data can be loaded into the application using the ***load*** command. At the moment only the ASF/AMC file format is supported but other file formats could be added easily. A list of all available input formats can be obtained by typing

```
list import_filters
```

at the console prompt.

The Acclaim ASF/AMC file format always consists of two files. The ****.ASF*** file contains information about the skeleton the motion is based on. The ****.AMC*** file contains the actual motion capture data.

Let's assume that there are two files, ***skeleton.asf*** and ***motion.amc***. To load the motion capture data the skeleton file has to be loaded first into the object repository. So typing

```
load s1 skeleton.asf
```

will load the skeleton from the file ***skeleton.asf*** and store it in the object repository using the name ***s1***. If an object with the specified name does already exist in the object repository it's contents will be overwritten.

Now the motion capture data can be loaded into the skeleton. To accomplish this

```
load s1 motion.amc
```

must by typed at the console prompt. This will load the motion stored in the ***motion.amc*** file into the skeleton ***s1*** stored in the object repository. If some other motion has already been loaded into the skeleton, it will be replaced by the new one.

Both steps can be combined, so typing

```
load s1 skeleton.asf motion.amc
```

will first load the skeleton file and store it in the object repository using the name ***s1***. Next the motion from the ***motion.amc*** file will automatically be loaded into the same skeleton object.

5.2 Visualizing motion capture data

The motion capture data loaded into the object repository can also be visualized. This can be done by using the ***visualize*** command. Let's assume that a skeleton and motion data have been loaded into an object ***s1*** as described in the section before. So typing

```
visualize s1
```

at the console prompt, will visualize the motion capture data. A new sub window will be displayed at the top area of the application's main window and the button *"**s1**"* will appear in the window button bar. The same data can be visualized several times. So typing

```
visualize s1
```

one more time will open a second visualization window. It is also possible to visualize several motions in the same window. Let's assume there are two objects ***s1*** and ***s2*** containing different motion capture data. So typing

```
visualize s1 s2
```

at the console prompt will visualize both motions in the same window simultaneously. This feature can for example be used to visually compare two motions. Theoretically an infinite number of motions can be visualized at the same time.

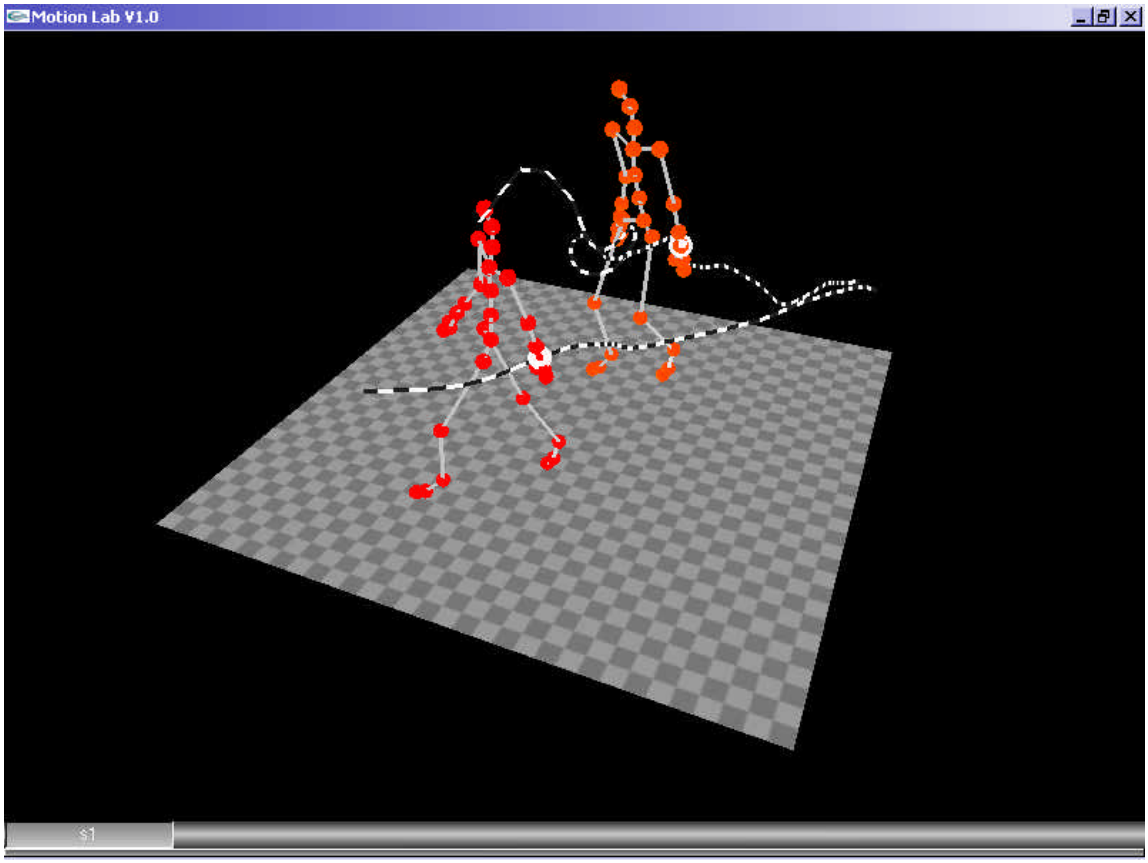

Figure 2 - Simultaneous visualization of two motions with activated paths

5.2.1 Controlling the visualization

It is possible to navigate through the displayed scene using the mouse. Holding the ***left mouse button*** and moving the mouse horizontally or vertically will rotate the camera around the center of the scene. Holding the ***middle mouse button*** and moving the mouse up or down will zoom the camera in or out.

Additionally the following keys are supported:

Key	*Description*
F2	The F2 key can be used to activate the animation. It toggles between the following playback states: no playback, normal playback, reverse playback
F1	The F1 key can be used to switch automatic camera movement on or of. If the automatic camera movement is activated the camera follows the motion of a skeleton. If more than one skeletons are visualized simultaneously the F1 key toggles between them.
Cursor left / Cursor right	The horizontal cursor keys can be used to manually switch between successive animation frames. The cursor left key jumps to the previous frame, while the cursor right key jumps to the next frame. This works only when animation playback is deactivated.
Cursor up / Cursor down	The vertical cursor keys can be used to change animation playback speed. The cursor up key increases the playback speed, while the cursor down key decreases it. This works only when animation playback is activated.
o / a	The o/a keys can be used to activate motion path visualization. When activated the motion path of a skeleton's node is shown as a dotted curve. The o/a keys can then be used to switch between nodes.

Table 2 - Keys supported by the motion capture visualization

Another possibility is to open the visualization pop up menu by clicking the ***right mouse button*** inside the visualization window. The pop up menu contains entries for each of the options mentioned above. Additionally a sub menu for each visualization plug in exists containing at least the *"Enabled/Disabled"* entry which can be used to activate and deactivate the plug in.

6 Calculating mass properties

For some types of motion analysis like for example inverse dynamics mass properties must be known for each bone of a skeleton. The *MotionLab* application provides an extensible and flexible mechanism to accomplish this task.

6.1 Mass calculation algorithms

The central idea of the mass calculation module is the so called mass calculation algorithm. Such an algorithm is responsible for the calculation of the mass properties for a certain skeleton model on which the computation is based. Each algorithm exposes a certain number of skeleton segments (or simply bones) for which mass properties (i.e. mass, moment of inertia, center of mass) are calculated. It is also possible to pass numerical parameters to an algorithm to control the computational process.

6.1.1 Listing available mass calculation algorithms

To list all mass calculation algorithms available to the application the ***list*** command can be used. Typing

```
list mass_algorithms
```

at the console prompt will output a list of available algorithms. At the moment only the ***Simple*** algorithm is available. It is a rudimentary algorithm for the calculation of the mass properties for the skeleton of a human. This algorithm is based on the paper *"Development of a computer tool for anthropometric analyses"* by *K.L.Robbins* and *Q.Wu*. The algorithm supports the following segments (bones) and the following mass properties:

- **head** : mass, moment of inertia
- **torso** : mass, moment of inertia
- **upper arm** : mass, moment of inertia, center of mass, length
- **lower arm** : mass, moment of inertia, center of mass, length
- **hand** : mass, moment of inertia, center of mass
- **thigh** : mass, moment of inertia, center of mass, length
- **shank** : mass, moment of inertia, center of mass, length
- **foot** : mass, moment of inertia, center of mass

Two parameters can be provided to the algorithm, the total body height and the total body mass.

6.1.2 Retrieving information about a mass calculation algorithm

It is also possible to get some detailed information about a certain algorithm. For this task the ***describe_algorithm*** command can be used followed by the name of an available algorithm. Let's assume that there is an algorithm called ***Simple***, so typing

```
describe_algorithm Simple
```

will output detailed information about the algorithm, like a list of segments for which mass properties are computed and a list of parameters which can be passed to the algorithm.

6.2 Mass assignment scripts

Since the segments exposed by an algorithm can be different to the segments of a motion capture skeleton there is a second mechanism provided called a mass assignment script. So the task of the such a script is to assign the data computed by an mass calculation algorithm to the nodes (bones) of a motion capture skeleton.

A mass assignment script simply consists of lines. Each line can be a comment or contain exactly one command which can be optionally followed by one or more arguments. The commands are executed sequentially, but also loops and sub procedure calls are supported. The commands can manipulate an operand stack and a symbol table. The symbol table contains data entries associated with a name, so it is simply a list of variables which can be accessed by the script commands.

There are various commands which can be used inside a script. The most basic commands are commands which are used to assign and move data between the operand stack and the symbol table. Also basic arithmetic commands and some basic vector / matrix operations are provided. At last there are some rudimentary control flow commands like compares, jumps and sub procedure calls. For a detailed description of the available commands read the mass assignment script command reference.

6.3 Computing mass properties

To compute the mass properties of a skeleton the ***calculate_mass*** command can be used. The first parameter of the command must be the name of a skeleton object contained inside the object repository. The second parameter must be the name of script file which will assign the computed mass properties to the skeleton's nodes. The third optional parameter is the name of a mass calculation algorithm, which should be executed before the script. The algorithm parameter can be followed by zero or more numerical parameters which will be passed to the algorithm itself.

Let's assume there is a skeleton object with the name ***s1***, a script file called ***script.txt*** and an algorithm called ***Simple*** which requires two parameters, the total body mass and the total body height. So typing

```
calculate_mass s1 script.txt Simple 75.0 180.0
```

at the console prompt will first execute the ***Simple*** algorithm and pass the values 75.0 (body mass in kg) and 180.0 (body height in cm) to it. After the algorithm has been executed the symbol table of the script interpreter is initialized with the computed values using a special naming convention. This naming convention is:

Symbol name	Description
`alg.segment_name.Mass`	Mass property of segment *segment_name*
`alg.segment_name.CenterOfMass`	Center of mass property of segment *segment_name*
`alg.segment_name.MomentOfInertia`	Moment of inertia property of segment *segment_name*
`alg.segment_name.Length`	Length property of segment *segment_name*

Table 3 - Symbol naming convention for segment mass properties

In the next step the mass assignment script ***script.txt*** is executed. It's task is to assign the calculated properties to the specified skeleton's bones. This is done by storing values inside the symbol table again using a special name convention. This naming convention is:

Symbol name	Description
`skl.bone_name.Mass`	Mass property of bone *bone_name*
`skl.bone_name.CenterOfMass`	Center of mass property of bone *bone_name*
`skl.bone_name.MomentOfInertia`	Moment of inertia property of bone *bone_name*
`skl.bone_name.Length`	Length property of bone *bone_name*

Table 4 - Symbol naming convention for skeleton bone mass properties

The part designated by ***bone_name*** must of course be replaced by the name of the bone inside the skeleton to which the properties should be assigned.

In the last step the values from the symbol table are taken and copied into the specified skeleton ***s1***.

Example:

```
set skl.head.Mass alg.head.Mass
```

This script statement would assign the value of the mass calculated for the *head* segment by the mass calculation algorithm to the motion capture skeleton's node *head*.

6.4 Script operation reference

This section gives an overview about the operations provided by the script interpreter used by the mass calculation module. Since many operations can be called in more than one way instead of a formal parameter description examples are given and their effects are explained.

6.4.1 The „pop“ operation

The ***pop*** operation removes one element from the top of the operand stack. If an integer constant is specified as an argument more than one elements can be removed simultaneously from the stack.

Example:

```
pop
```

This pop one element from the stack. If the stack is empty a stack underflow error is raised.

Example:

```
pop 5
```

This pops exactly 5 elements from the stack. If the stack size is smaller than the specified number of elements to pop, a stack underflow error is raised.

6.4.2 The „push_int“ operation

The ***push_int*** operation pushes one or more integer values onto the operand stack.

Example:

```
push_int 102
```

Pushes the integer value of 102 onto the operand stack.

Example:

```
push_int 106 120 -524
```

Pushes the integer values 106, 120 and 524 onto the operand stack in the given order.

6.4.3 The „push_real“ operation

The ***push_real*** operation pushes one or more real values onto the operand stack.

Example:

```
push_int 0.5
```

Pushes the real value of 0.5 onto the operand stack.

Example:

```
push_int 0.35 10.4 -8.65
```

Pushes the real values 0.35, 10.4 and -8.65 onto the operand stack in the given order.

6.4.4 The „push_string“ operation

The ***push_string*** operation converts all arguments into strings and pushes them onto the operand stack.

Example:

```
push_string “Hello”
```

Pushes the string literal “Hello” onto the operand stack.

Example:

```
push_string “Hello” “World”
```

Pushes the string literals “Hello” and “World” onto the operand stack in the given order.

Example:

```
push_string 16.42
```

Converts the number 16.42 into a string and pushes the string onto the operand stack.

6.4.5 The „push_vector“ operation

The ***push_vector*** operation pushes a vector onto the operand stack.

Example:

```
push_vector 0 1 0
```

Pushes the vector (0,1,0) onto the operand stack.

6.4.6 The „push_matrix“ operation

The ***push_matrix*** operation pushes a matrix onto the operand stack. The rows of the matrix have to be separated using the **“|”** character. Missing components are assumed to be zero. The matrix has the width of longest row.

Example:

```
push_matrix 1 0 | 0 1
```

Pushes the matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ onto the operand stack.

Example:

```
push_matrix 1 | 2 | 3 0 0
```

Pushes the matrix $\begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}$ onto the operand stack.

6.4.7 The „set“ operation

The ***set*** operation assigns either the top most element of the stack to a symbol inside the symbol table or it copies the value of a symbol into an other symbol. In the first case the first argument must be an identifier specifying the name of the destination symbol. In the second case the first argument must be an identifier specifying the name of the destination symbol and the second argument must be an identifier specifying the name of the source symbol. The destination symbol is automatically created. If the source symbol cannot be found an error is raised.

Example:

```
set destination
```

The value of the topmost operand on the operand stack is taken and assigned to the symbol *destination*. The operand remains on the stack.

Example:

```
set destination source
```

The value of the symbol *source* is assigned to the symbol *destination.*

6.4.8 The „set_int“ operation

The ***set_int*** operation either assigns the value of the top most element of the stack to a symbol inside the symbol table or it assigns the value of an integer literal to a symbol. In the first case the first argument must be an identifier specifying the name of the destination symbol. The value which should be assigned is taken from the operand stack but must be an integer. If the operand is not an integer an invalid operand type error is raised. In the second case the first argument must be an identifier specifying the name of the destination symbol and the second argument must be an integer value which should be assigned to the destination symbol. The destination symbol is automatically created.

Example:

```
set destination
```

The value of the topmost operand on the operand stack is taken and assigned to the symbol *destination*. The operand must be an integer value and remains on the stack.

Example:

```
set destination 5
```

The integer value 5 is assigned to the symbol *destination.* After the operation has been executed the symbol *destination* is an integer.

6.4.9 The „set_real“ operation

The ***set_real*** operation either assigns the value of the top most element of the stack to a symbol inside the symbol table or it assigns the value of an integer or an real literal to a symbol. In the first case the first argument must be an identifier specifying the name of the destination symbol. The value which should be assigned is taken from the operand stack but must be an integer or an real. If the operand is neither an integer nor a real an invalid operand type error is raised. In the second case the first argument must be an identifier specifying the name of the destination symbol and the second argument must be an integer or a real value which should be assigned to the destination symbol. The destination symbol is automatically created.

Example:

```
set destination
```

The value of the topmost operand on the operand stack is taken and assigned to the symbol *destination*. The operand must be a real or an integer value and remains on the stack.

Example:

```
set destination 5.52
```

The real value 5.52 is assigned to the symbol *destination.* After the operation has been executed the symbol *destination* is a real.

6.4.10 The „set_string“ operation

The ***set_string*** operation either assigns the value of the top most element of the stack to a symbol inside the symbol table or it assigns the value of the second argument to a symbol. In the first case the first argument must be an identifier specifying the name of the destination symbol. The value which should be assigned is taken from the operand stack but must be a string. If the operand is not a string an invalid operand type error is raised. In the second case the first argument must be an identifier specifying the name of the destination symbol and the second argument is converted into a string and assigned to the destination symbol. The destination symbol is automatically created.

Example:

```
set destination
```

The value of the topmost operand on the operand stack is taken and assigned to the symbol *destination*. The operand must be a string and remains on the stack.

Example:

```
set destination “Hello World”
```

The string literal “Hello World” is assigned to the symbol *destination.* After the operation has been executed the symbol *destination* is a string.

Example:

```
set destination 55.6123
```

The integer value 55.6123 is converted into a string and assigned to the symbol *destination.* After the operation has been executed the symbol *destination* is a string.

6.4.11 The „set_vector“ operation

The ***set_vector*** operation either assigns the value of the top most element of the stack to a symbol inside the symbol table or it assigns the vector build up from the remaining arguments which must be numerical to a symbol. In the first case the first argument must be an identifier specifying the name of the destination symbol. The value which should be assigned is taken from the operand stack but must be a vector. If the operand is not a vector an invalid operand type error is raised. In the second case the first argument must be an identifier specifying the name of the destination symbol. The first argument is then followed by one or more numerical values which are the components of the vector which should be assigned to the destination symbol. The destination symbol is automatically created.

Example:

```
set destination
```

The value of the topmost operand on the operand stack is taken and assigned to the symbol *destination*. The operand must be a vector and remains on the stack.

Example:

```
set destination 0 1 0
```

The vector (0,1,0) is assigned to the symbol *destination.* After the operation has been executed the symbol *destination* is a vector.

6.4.12 The „set_matrix“ operation

The ***set_matrix*** operation either assigns the value of the top most element of the stack to a symbol inside the symbol table or it assigns the matrix build up from the remaining arguments, which must be numerical or the row delimiter “|”, to a symbol. In the first case the first argument must be an identifier specifying the name of the destination symbol. The value which should be assigned is taken from the operand stack but must be a matrix. If the operand is not a matrix an invalid operand type error is raised. In the second case the first argument must be an identifier specifying the name of the destination symbol. The first argument is then followed by one or more numerical values or the row delimiter “|” from which the components of the matrix which should be assigned to the destination symbol will be build. The destination symbol is automatically created.

Example:

```
set destination
```

The value of the topmost operand on the operand stack is taken and assigned to the symbol *destination*. The operand must be a matrix and remains on the stack.

Example:

```
set destination 1 0 0 | 0 1 0 | 0 0 1
```

The matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ is assigned to the symbol *destination.* After the operation has been executed the symbol *destination* is a matrix.

6.4.13 The „get“ operation

The ***get*** operation pushes the value of a symbol stored inside the symbol table onto the stack. The operation expects exactly one argument which must be an identifier specifying the name of the symbol from which the value should be taken.

Example:

```
get source
```

Pushes the value stored inside the symbol *source* onto the operand stack.

6.4.14 The „get_int“ operation

The ***get_int*** operation pushes the value of a symbol stored inside the symbol table onto the stack. The operation expects exactly one argument which must be an identifier specifying the name of the symbol from which the value should be taken. The data type of the symbol must be integer or real. A real value is converted automatically into an integer. If the type is neither integer nor real an invalid operand type error is raised.

Example:

```
get source
```

Pushes the value stored inside the symbol *source* onto the operand stack. The value must be an integer or a real. The type of the operand pushed onto the stack is integer.

6.4.15 The „get_real“ operation

The ***get_real*** operation pushes the value of a symbol stored inside the symbol table onto the stack. The operation expects exactly one argument which must be an identifier specifying the name of the symbol from which the value should be taken. The data type of the symbol must be integer or real. An integer value is converted automatically into a real. If the type is neither integer nor real an invalid operand type error is raised.

Example:

```
get source
```

Pushes the value stored inside the symbol *source* onto the operand stack. The value must be an integer or a real. The type of the operand pushed onto the stack is real.

6.4.16 The „get_string“ operation

The ***get_string*** operation pushes the value of a symbol stored inside the symbol table onto the stack. The operation expects exactly one argument which must be an identifier specifying the name of the symbol from which the value should be taken. The data type of the symbol must be string otherwise an invalid operand type error is raised.

Example:

```
get source
```

Pushes the value stored inside the symbol *source* onto the operand stack. The value must be a string. The type of the operand pushed onto the stack is string.

6.4.17 The „get_vector“ operation

The ***get_vector*** operation pushes the value of a symbol stored inside the symbol table onto the stack. The operation expects exactly one argument which must be an identifier specifying the name of the symbol from which the value should be taken. The data type of the symbol must be vector otherwise an invalid operand type error is raised.

Example:

```
get source
```

Pushes the value stored inside the symbol *source* onto the operand stack. The value must be a vector. The type of the operand pushed onto the stack is vector.

6.4.18 The „get_matrix“ operation

The ***get_matrix*** operation pushes the value of a symbol stored inside the symbol table onto the stack. The operation expects exactly one argument which must be an identifier specifying the name of the symbol from which the value should be taken. The data type of the symbol must be matrix otherwise an invalid operand type error is raised.

Example:

```
get source
```

Pushes the value stored inside the symbol *source* onto the operand stack. The value must be a matrix. The type of the operand pushed onto the stack is matrix.

6.4.19 The „add“ operation

The ***add*** operation computes the sum of two operands and pushes it onto the stack. It expects either zero, one or two arguments. In the first case both operands are removed from the stack and the result of their sum is pushed onto the stack. In the second case the first operand is taken from the stack the second operand is taken from the symbol table, thus the argument of the ***add*** operation must be an identifier specifying the symbol. In the third case both operands are taken from the symbol table. Their names must be specified by the arguments passed to the ***add*** operation. The following combinations of operand types are allowed:

Operand A	***Operand B***	***A+B***
Integer	Integer	Integer
Integer	Real	Real
Real	Integer	Real
Vector	Vector	Vector
Matrix	Matrix	Matrix

*Table 5 - Valid operand combinations for the **add** operation*

If vectors or matrices are added their size must match otherwise an operands don't match error is raised.

Example:

```
add
```

Takes both operands from the stack, removes them and pushes their sum onto the stack.

Example:

```
add b
```

Takes the first operand from the stack, removes it, takes the second operand from the symbol *b* and pushes their sum onto the stack.

Example:

```
add a b
```

Takes the value of the first operand from the symbol *a*, takes the value of the second operand from the symbol *b* and pushes their sum onto the stack.

6.4.20 The „sub“ operation

The ***sub*** operation computes the difference of two operands and pushes it onto the stack. It expects either zero, one or two arguments. In the first case both operands are removed from the stack, whereby the topmost operand is the one which is subtracted from the operand below, and the result of their difference is pushed onto the stack. In the second case the first operand is taken from the stack the second operand is taken from the symbol table, thus the argument of the ***sub*** operation must be an identifier specifying the symbol. In the third case both operands are taken from the symbol table. Their names must be specified by the arguments passed to the ***sub*** operation. The following combinations of operand types are allowed:

Operand A	***Operand B***	***A-B***
Integer	Integer	Integer
Integer	Real	Real
Real	Integer	Real
Vector	Vector	Vector
Matrix	Matrix	Matrix

*Table 6 - Valid operand combinations for the **sub** operation*

If vectors or matrices are subtracted their size must match otherwise an operands don't match error is raised.

Example:

```
sub
```

Takes both operands from the stack, removes them and pushes their difference onto the stack. The topmost operand is subtracted from the operand below.

Example:

```
sub b
```

Takes the first operand from the stack, removes it, takes the second operand from the symbol *b* and pushes their difference onto the stack.

Example:

```
sub a b
```

Takes the value of the first operand from the symbol *a*, takes the value of the second operand from the symbol *b* and pushes their difference onto the stack.

6.4.21 The „mul“ operation

The ***mul*** operation computes the product of two operands and pushes it onto the stack. It expects either zero, one or two arguments. In the first case both operands are removed from the stack and the result of their product is pushed onto the stack. In the second case the first operand is taken from the stack the second operand is taken from the symbol table, thus the argument of the ***mul*** operation must be an identifier specifying the symbol. In the third case both operands are taken from the symbol table. Their names must be specified by the arguments passed to the ***mul*** operation. The following combinations of operand types are allowed:

Operand A	***Operand B***	***A*B***
Integer	Integer	Integer
Integer	Real	Real
Real	Integer	Real
Vector	Integer	Vector
Vector	Real	Vector
Integer	Vector	Vector
Real	Vector	Vector
Vector	Vector	Real
Integer	Matrix	Matrix
Real	Matrix	Matrix
Matrix	Integer	Matrix
Matrix	Real	Matrix
Matrix	Matrix	Matrix
Vector	Matrix	Vector
Matrix	Vector	Matrix

*Table 7 - Valid operand combinations for the **mul** operation*

If vectors or matrices are multiplied their size must match otherwise an operands don't match error is raised. When vectors are multiplied their dot product is calculated. Multiplying a scalar with a vector or a matrix multiplies each component of the vector or the matrix with the scalar.

Example:

```
mul
```

Takes both operands from the stack, removes them and pushes their product onto the stack.

Example:

```
mul b
```

Takes the first operand from the stack, removes it, takes the second operand from the symbol *b* and pushes their product onto the stack.

Example:

```
mul a b
```

Takes the value of the first operand from the symbol *a*, takes the value of the second operand from the symbol *b* and pushes their product onto the stack.

6.4.22 The „div“ operation

The ***div*** operation computes the quotient of two operands and pushes it onto the stack. It expects either zero, one or two arguments. In the first case both operands are removed from the stack, whereby the topmost operand is the one which by which the operand below is divided, and the result of their quotient is pushed onto the stack. In the second case the first operand is taken from the stack the second operand is taken from the symbol table, thus the argument of the ***div*** operation must be an identifier specifying the symbol. In the third case both operands are taken from the symbol table. Their names must be specified by the arguments passed to the ***div*** operation. The following combinations of operand types are allowed:

Operand A	***Operand B***	***A/B***
Integer	Integer	Integer
Integer	Real	Real
Real	Integer	Real
Vector	Integer	Vector
Vector	Real	Vector
Matrix	Integer	Matrix
Matrix	Real	Matrix

*Table 8 - Valid operand combinations for the **div** operation*

If the second operand is zero a division by zero error is raised. When a matrix or a vector is divided by a scalar each of their components is divided by the scalar value.

Example:

```
div
```

Takes both operands from the stack, removes them and pushes their quotient onto the stack. The topmost operand is the one by which the the operand below is divided.

Example:

```
sub b
```

Takes the first operand from the stack, removes it, takes the second operand from the symbol *b* and pushes their quotient onto the stack.

Example:

```
sub a b
```

Takes the value of the first operand from the symbol *a*, takes the value of the second operand from the symbol *b* and pushes their quotient onto the stack.

6.4.23 The „vector_product“ operation

The ***vector_product*** operation computes the vector products of two operands and pushes it onto the stack. The operation expects both operands to be a three dimensional vector. It expects either zero, one or two arguments. In the first case both operands are removed from the stack and the result of their vector product is pushed onto the stack. In the second case the first operand is taken from the stack the second operand is taken from the symbol table, thus the argument of the ***vector_product*** operation must be an identifier specifying the symbol. In the third case both operands are taken from the symbol table. Their names must be specified by the arguments passed to the ***vector_product*** operation. If the types of the arguments are not vectors or the vectors are not three dimensional a operands don't match error is raised.

Example:

```
vector_product
```

Takes both operands from the stack, removes them and pushes their product onto the stack. The operands must be three dimensional vectors.

Example:

```
vector_product b
```

Takes the first operand from the stack, removes it, takes the second operand from the symbol *b* and pushes their vector product onto the stack. The operands must be three dimensional vectors.

Example:

```
vector_product a b
```

Takes the value of the first operand from the symbol *a*, takes the value of the second operand from the symbol *b* and pushes their vector product onto the stack. The operands must be three dimensional vectors.

6.4.24 The „vector_length“ operation

The ***vector_length*** operation computes the euclidean length of a vector and pushes it onto the operand stack. It expects either zero or one argument. In the first case the vector whose length should be computed must be on the top of the stack whereby the vector itself is not removed from the stack. In the second case the argument must by an identifier specifying the name of a symbol which contains the vector whose length should be calculated. If the operand is not a vector an invalid operand type error is raised.

Example:

```
vector_length
```

The topmost operand on the stack must be a vector. It's length is calculated and pushed onto the stack. The vector is not removed.

Example:

```
vector_length v
```

The vector is taken from the symbol *v*. It's computed length is pushed onto the stack.

6.4.25 The „transpose“ operation

The ***transpose*** operation calculates the transpose of a matrix and pushes it onto the operand stack. It expects zero or one arguments. In the first case the matrix whose transpose should be computed is taken from the operand stack but not removed from it. In the second case the argument must be an identifier specifying the name of a symbol containing the matrix whose transpose should be computed. If the operand is not a matrix an invalid operand type error is raised.

Example:

```
transpose
```

The topmost operand on the stack must be a matrix. It's transpose is calculated and pushed onto the stack. The matrix is not removed.

Example:

```
transpose v
```

The matrix is taken from the symbol *v*. It's computed transpose is pushed onto the stack.

6.4.26 The „get_vector_component“ operation

The ***get_vector_component*** fetches the component of a vector specified by a zero based index and pushes the value of the component onto the operand stack. The operation expects zero, one or two arguments.

If no arguments are passed the operation expects the top most stack operand to be an integer value specifying the index of the component which should be returned. The operand below must be a vector from which the component should be read.

If one argument is passed the argument must be either an integer value or an identifier specifying the name of a symbol containing the an zero based integer index of the component which should be returned. If the symbol does not exist a symbol not found error is raised. If the type of the symbol is not integer an invalid operand type error is raised. The topmost operand of the stack is expected to be the vector from which the component should be read. If the operand is not a vector an invalid operand type error is raised.

If two arguments are passed the first argument must be an identifier specifying the name of a symbol containing a vector from which the component should be read. If the symbol does not exist a symbol not found error is raised. If the symbol is not a vector an invalid operand type error is raised. The second argument is expected to be either an integer value or an identifier specifying a symbol containing an zero based integer index of the component which should be returned. If the symbol does not exist a symbol not found error is raised. If the symbol is not an integer an invalid argument type error is raised.

If the specified index is out of range the operation raises an out of range error. None of the operands is removed from the stack. The value of the specified component is simply pushed onto the stack.

Example:

```
get_vector_component
```

Returns the component specified by the topmost operand on the stack of the vector specified by the component below.

Example:

```
get_vector_component 3
```

Returns the fourth (the first component has an index of zero) component of the vector which must be the topmost operand on the stack.

Example:

```
get_vector_component index
```

Returns the component whose index is specified by the integer symbol *index*. The topmost operand on the stack must be the vector form which the component should be read.

Example:

```
get_vector_component vector 2
```

Returns the third (the first component has an index of zero) component of the vector contained inside the symbol *vector*.

Example:

```
get_vector_component vector index
```

Returns the component whose index is specified by the integer symbol *index* from the vector which is stored inside the symbol *vector*.

6.4.27 The „get_matrix_component“ operation

The ***get_matrix_component*** operation fetches the component of a matrix specified by a zero based row/col coordinate pair. The value of the component is pushed onto the stack. The operation expects zero,one, two or three arguments.

If no arguments are passed the operands are fetched from the stack. Let *st(0)* be the topmost operand of the stack, *st(1)* the operand below and so on. In this case *st(0)* is expected to be the zero based integer index of the column from which the component should be fetched, *st(1)* is expected to be the zero based integer index of the row from which the component should be fetched and at last *st(2)* is expected to be the matrix from which the component should be read. None of the operands is removed from stack, the value of the component is simply pushed onto the stack. If the types of the operand are invalid and invalid operand type error is raised.

If one argument is passed it is expected to be an identifier specifying a a symbol containing the matrix from which the component should be read. If the specified symbol does not exist a symbol not found error is raised. If the symbol is not a matrix an invalid operand type error is raised. The topmost operand on the stack is expected to be the zero based integer index of the column from which the component should be read and the operand below must be the zero based integer index of the row from which the component should be fetched. If both operands are not integer values an invalid operand type error is raised.

If two arguments are passed they might either be integer values or identifiers specifying symbols containing an integer value. If the symbols do not exists a symbol not found error is raised. If the symbols are not integers an invalid operand type is raised. The first argument is expected to be the zero based integer index of the row from which the component should be fetched. The second argument is expected to be the zero based integer index of the column from which the component should be fetched. The top most operand on the operand stack must be a matrix. From this matrix the value of the specified component is read and pushed onto the stack.

If three arguments are specified the first argument must be an identifier specifying a symbol which contains the matrix from which the component should be fetched. The two following arguments can either be an integer value or an identifier specifying a symbol which contains an integer value. The second argument is expected to be the zero based integer index of the row from which the component should be fetched. The third argument is expected to be the zero based integer index of the column from which the component should be read.

If the specified coordinates of the component which should be fetched are out of range an out of range error is raised. If the stack does not contain enough elements a stack underflow error is raised.

Example:

```
get_matrix_component
```

All operands must be on the stack. The topmost operand st(0) must be the column index, the operand st(1) the row index and the operand st(2) the matrix from which the specified component should be read and pushed onto the stack.

Example:

```
get_matrix_component matrix
```

The topmost operand on the stack is expected to be the column index, the operand below the row index. The component is fetched from the matrix contained in the symbol *matrix*.

Example:

```
get_matrix 0 0
```

Fetches the top left component of the matrix which must be on the top of the operand stack.

Example:

```
get_matrix row col
```

Fetches the component whose coordinates are specified by the symbols *row* and col from the matrix which must be on the top of the stack.

Example:

```
get_matrix matrix 0 0
```

Fetches the top left component of the matrix contained in the symbol *matrix*.

Example:

```
get_matrix matrix row col
```

Fetches the component whose coordinates are specified by the symbols *row* and *col* from the matrix contained in the symbol *matrix*.

6.4.28 The „set_vector_component“ operation

The ***set_vector_component*** operation sets the value of a vector's component. The operation expects zero, two or three arguments.

If no arguments are passed all operands are expected to be on the operand stack. Let *st(0)* be the topmost operand, *st(1)* the operand below and so on. In this case *st(0)* is expected to be a real or integer value which should be written to the vector. The operand st(1) is expected to be the zero based integer index of the component to which the value should be written. The operand st(2) must be the vector whose component should be set. If the types of the operands are invalid an invalid operand type error is raised. If the stack does not contain enough elements a stack underflow error is raised.

If one or two arguments are passed they are expected to be either numerical literals or identifiers specifying symbols. If three arguments are passed the first argument must be a symbol identifier the two remaining arguments might be either numerical literals or symbol identifiers.

The following combinations of operands are allowed:

arg(0)	***arg(1)***	***arg(2)***	***st(0)***	***st(1)***	***st(2)***
-	-	-	**Component value** real or integer	**Index** real/integer	**Vector** vector
Vector symbol identifier	-	-	**Component value** real/integer	**Index** real/integer	-
Component value symbol identifier/numerical literal	-	-	**Index** real/integer	**Component value** real/integer	-
Index symbol identifier/numerical literal	**Component value** symbol identifier/numerical literal	-	**Vector** vector	-	-
Vector symbol identifier	**Index** symbol identifier/numerical literal	-	**Component value** rea/integer	-	-
Vector symbol identifier	**Index** symbol identifier/numerical literal	**Component value** symbol identifier/numerical literal	-	-	-

*Table 9 - Valid operand combinations for the **set_vector_component** operation*

If the specified component index is out of range an out of range error is raised. If the operand types are invalid an invalid operand type error is raised. None of the operands is removed from stack.

6.4.29 The „set_matrix_component“ operation

The ***set_matrix_component*** operation sets the value of a matrix component. The operation expects zero, two, three or four arguments.

Let *st(0)* be the topmost operand, *st(1)* the operand below, *arg(0)* be the first argument, *arg(1)* the second and so on. The the following operand combinations are allowed:

arg(0)	*arg(1)*	*arg(2)*	*arg(3)*	*st(0)*	*st(1)*	*st(2)*	*st(3)*
-	-	-	-	**Value** integer/real	**Column** integer/real	**Row** integer/real	**Matrix** matrix
Matrix symbol identifier	-	-	-	**Value** integer/real	**Column** integer/real	**Row** integer/real	-
Value symbol identifier/ numerical literal	-	-	-	**Column** integer/real	**Row** integer/real	**Matrix** matrix	-
Row symbol identifier/ numerical literal	**Column** symbol identifier/ numerical literal	-	-	**Value** integer/real	**Matrix** matrix	-	-
Matrix symbol identifier	**Value** symbol identifier/ numerical literal	-	-	**Row** integer/real	**Column** integer/real	-	-
Matrix symbol identifier	**Row** symbol identifier/ numerical literal	**Column** symbol identifier/ numerical literal	**-**	**Value** integer/real	-	-	-
Row symbol identifier/ numerical literal	**Column** symbol identifier/ numerical literal	**Value** symbol identifier/ numerical literal	**-**	**Matrix** matrix	-	-	**-**
Matrix symbol identifier	**Row** symbol identifier/ numerical literal	**Column** symbol identifier/ numerical literal	**Value** symbol identifier/ numerical literal	**-**	**-**	**-**	**-**

*Table 10 - Valid operand combinations for the **set_matrix_component** operation*

The row and column indices are zero based, so the top left component has the coordinate (0,0). If the specified component coordinates are out of range an out of range error is raised. If the operand types are invalid an invalid operand type error is raised. None of the operands is removed from stack.

6.4.30 The „cmp“ operation

The ***cmp*** operation compares two operands. If both operands are equal zero is pushed onto the operand stack. If the second operand is greater than the first a value greater zero is pushed onto the stack. If the second operand is smaller than the first a value less zero is pushed onto the stack. The operands must have the same type otherwise an operands don't match error is raised. The only exception is the comparison of real and integer values. The integer values will be converted into reals automatically. When vectors or matrices are compared simply their components are compared using an epsilon interval. Two vectors are equal if the absolute distance between each component pair is less equal than the epsilon value, otherwise the vectors are unequal. Two matrices are equal if the absolute distance between each component pair is less equal than the epsilon value, otherwise the matrices are unequal.

The ***cmp*** operation expects either zero, one or two arguments. In the first case both operands are taken from the stack whereby the topmost operand is the second one the operand below the first one. In the second case the first operand is taken from the stack. The second operand is either a numeric literal, a string literal or an identifier which specifies the name of a symbol whose value is taken as the second operand. In the third case the first argument is either a literal, a string literal or an identifier. The same applies for the second argument. If the arguments are identifiers the operand values are taken from the symbol table. If a symbol cannot be found a symbol not found is raised. If the stack does not contain enough elements then a stack underflow error is raised.

Example:

```
cmp
```

Both operands must lie on the operand stack. They are compared an the result of the comparison is pushed onto the stack. The operands are not removed by the ***cmp*** operation.

Example:

```
cmp 5
```

The first operand must lie on the operand stack. It is compared with the value 5. In this case first operand must either be an integer or a real. The result of the comparison is pushed onto the stack.

Example:

```
cmp “Hello World!”
```

The first operand must lie on the operand stack. It is compared with the string literal “Hello World!” In this case first operand must be a string. The result of the comparison is pushed onto the stack.

Example:

```
cmp second
```

The first operand must lie on the operand stack. It is compared with the value of the symbol *second* contained in the symbol table. In this case both operands must have the same type. The result of the comparison is pushed onto the stack.

Example:

```
cmp 10 5
```

The values 10 and 5 are compared. The result of the comparison is pushed onto the stack.

Example:

```
cmp first 5
```

The value of the symbol *first* is compared to 5. The result of the comparison is pushed onto the stack. The type of the symbol must be real or integer.

Example:

```
cmp first "Hello World"
```

The value of the symbol *first* is compared to the string literal "Hello World". The result of the comparison is pushed onto the stack. The type of the symbol must be string.

Example:

```
cmp first second
```

The values of the symbols *first* and *second* are compared. The result of the comparison is pushed onto the stack. The types of both symbols must be equal.

6.4.31 The „jmp" operation

The ***jmp*** operation performs an unconditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script.

Example:

```
jmp destination
```

Jumps to the label *destination*. In the next program cycle the first operation after the label is executed.

6.4.32 The „jmp_eq" operation

The ***jmp_eq*** operation performs a conditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The topmost value is removed from the stack and it's type must be integer. If the value is zero the jump is taken and the program execution is continued with the first operation following the label. Otherwise the jump is not taken and the execution is continued with the operation following the ***jmp_eq*** operation.

Example:

```
jmp_eq destination
```

Jumps to the label *destination* if the topmost stack operand is equal to zero. Otherwise the jump is not taken.

6.4.33 The „jmp_neq“ operation

The ***jmp_neq*** operation performs a conditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The topmost value is removed from the stack and it's type must be integer. If the value is non zero the jump is taken and the program execution is continued with the first operation following the label. Otherwise the jump is not taken and the execution is continued with the operation following the ***jmp_neq*** operation.

Example:

```
jmp_neq destination
```

Jumps to the label *destination* if the topmost stack operand is not zero. Otherwise the jump is not taken.

6.4.34 The „jmp_gr“ operation

The ***jmp_gr*** operation performs a conditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The topmost value is removed from the stack and it's type must be integer. If the value is greater than zero the jump is taken and the program execution is continued with the first operation following the label. Otherwise the jump is not taken and the execution is continued with the operation following the ***jmp_gr*** operation.

Example:

```
jmp_gr destination
```

Jumps to the label *destination* if the topmost stack operand is greater than zero. Otherwise the jump is not taken.

6.4.35 The „jmp_greq“ operation

The ***jmp_greq*** operation performs a conditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The topmost value is removed from the stack and it's type must be integer. If the value is greater than zero or zero the jump is taken and the program execution is continued with the first operation following the label. Otherwise the jump is not taken and the execution is continued with the operation following the ***jmp_greq*** operation.

Example:

```
jmp_greq destination
```

Jumps to the label *destination* if the topmost stack operand is greater than zero or zero. Otherwise the jump is not taken.

6.4.36 The „jmp_ls“ operation

The ***jmp_ls*** operation performs a conditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The topmost value is removed from the stack and it's type must be integer. If the value is smaller than zero the jump is taken and the program execution is continued with the first operation following the label. Otherwise the jump is not taken and the execution is continued with the operation following the ***jmp_ls*** operation.

Example:

```
jmp_ls destination
```

Jumps to the label *destination* if the topmost stack operand is smaller than zero. Otherwise the jump is not taken.

6.4.37 The „jmp_lseq“ operation

The ***jmp_lseq*** operation performs a conditional jump. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The topmost value is removed from the stack and it's type must be integer. If the value is smaller than zero or zero the jump is taken and the program execution is continued with the first operation following the label. Otherwise the jump is not taken and the execution is continued with the operation following the ***jmp_lseq*** operation.

Example:

```
jmp_ls destination
```

Jumps to the label *destination* if the topmost stack operand is smaller than zero or zero. Otherwise the jump is not taken.

6.4.38 The „call“ operation

The ***call*** operation performs a sub routine call. The operation expects exactly one argument which must be an identifier specifying the name of a valid label defined inside the script. The operation pushes the current value of the instruction pointer onto the stack and jumps to the position of the label. The program execution is continued with the first operation following the label.

Example:

```
call destination
```

Pushes the current value of the instruction pointer and jumps to the position designated by the label *destination*. The program execution is continued with the first operation following the label.

6.4.39 The „return“ operation

The ***return*** operation returns from a sub routine call. The operation expects no arguments. The top most value of the operand stack is assumed to be an integer value. If this is not the case an invalid operand type error is raised. If the stack is empty a stack underflow error is raised.

The integer value must be a valid position of the instruction pointer. The instruction pointer is set to this value by the operation. The program execution is continued with the operation following the operation to which the instruction pointer points.

Example:

```
return
```

Returns from a subroutine.

MotionLab

- Programmer's Guide -

Christoph Brzozowski
Stanley Klemme
Joachim Melzer
Hoang Nguyen

Contents

1 Overview

The *MotionLab* application was designed to be as easy expandable as possible. This document gives a short overview on how to add new functionality to the *MotionLab* framework at different levels. It is assumed, that the reader is familiar with the C/C++ language and object oriented programming. What this document will not explain, is how to implement the functionality itself. It will only show which mechanisms are provided to integrate new functionality into the framework.

2 Adding a new application module to the framework

The main part of the *MotionLab* framework resides inside the *CApplication* class. This class for example implements the basic behavior of the application's user interface, which includes a full console emulation using GLUT and OpenGL. It is responsible for some background management issues as well. All other specialized functionality is realized by plug ins, which are called ***application modules***. They are the entry point for extending the framework at the highest possible level.

An application module has to expose at least some information about itself, i.e. it's name and a basic description. Additionally, a set of commands can be implemented, which can be executed by the user through the application framework's console. More sophisticated application modules can create and add sub windows to the framework's user interface. This feature can be used for visualization purposes and to extend the user interface.

This section will lead you step by step through the process of implementing a new application module.

2.1 Step 1: Creating a new application module class

The very first step is creating a class for the new application module. This class has to derive from the abstract *IApplicationModule* interface class and implement all methods defined by this interface.

IApplicationModule
const std::string & getName() *const std::string & getDescription()* *int getCommandCount()* *const std::string & getCommand (int index)* *bool executeCommand(const std::string &command, const CCommandArgumentList &arguments)* *void initialize()* *void finalize()*

*Table 1- The **IApplicationModule** interface*

More detailed information about the *IApplicationModule* interface can be found inside the API documentation.

Example:

```
#include "IApplicationModule.h"
class CHelloWorldModule : public IApplicationModule
{
      // Here all methods defined by the IApplicationModule interface
      // must be overwritten
};
```

2.2 Step 2: Exposing basic module information

Each application module has to expose at least it's name and a short description. This is done by implementing the methods *getName()* and *getDescription()*. The first returns a reference to a string object containing the module's name, the latter returns a reference to a string object containing the module's description.

NOTE: The name of the module has to be unique.

Example:

```
const std::string& CHelloWorldModule::getName()
{
      static std::string Name = "HelloWorldModule";
      return Name
}

const std::string& CHelloWorldModule::getDescription()
{
      static std::string Description = "A simple HelloWorld! Module"
      return Description
}
```

2.3 Step 3: Exposing information about implemented commands

Each application module has the possibility of implementing a set of commands, which the user can call by entering them at the application framework's console. The module has to tell the framework, how many commands it implements and what their names are. These names should be unique and are handled case sensitive, i.e. *kill* and *KILL* will be handled as different commands. Please note, that the command name must not contain spaces.

This information can be exposed by implementing the methods *getCommandCount()* and *getCommand()*. The first method is meant to return the number of commands the module implements. If the module does not implement any command, this method should return zero. The second method is meant to return the name of the supported command designated by the *index* parameter. The first command has an index of zero, the last one an index of *getCommandCount()-1*. If the index is out of range, the method should return an empty string ("").

Example:

```
int CHelloWorldModule::getCommandCount()
{
      return 1;
}

const std::string& CHelloWorldModule::getCommand(int index)
{
      static std::string HelloWorldCommand = "hello_world"
      static std::string NULL_STR = "";
      if (index==0) return HelloWorldCommand; else return NULL_STR;
}
```

In this example the application module *CHelloWorldModule* implements only the command *hello_world*.

2.4 Step 4: Module initialization and finalization

Each application module will be initialized by the framework at application startup and finalized at application shutdown. To do so, the framework calls the *initialize()* method of each registered application module at application startup and the *finalize()* method at application shutdown. If the application module needs to initialize some dynamic resources at startup and release them later, it should implement both methods. Otherwise their implementation can be left empty.

Example:

```
void CHelloWorld::initialize()
{
      /// Empty since no initialization is needed
}

void CHelloWorld::finalize()
{
      /// Empty since no finalization is needed
}
```

In this example both implementations are left empty, because no initialization and finalization is required.

2.5 Step 5: Implementing commands

Before an application module's set of commands can be used by the framework, the commands have to be made accessible. This is done by implementing the *executeCommand()*. The name of the command to be executed and a list of the arguments to be passed to the command, are passed to this method. The method returns ***true*** if the command could be executed successfully. It returns ***false,*** if the execution of the command failed or the specified command is not supported by the application module.

The arguments passed to a command can be accessed using the *arguments* parameter. It has the type *std::vector<std::string>,* so the number of arguments passed to the command can be obtained by calling the vector's *size()* method. The values of the arguments can be accessed simply by using the array operator.

A possible implementation would be to define a protected method for each supported command and call this method from inside the *executeCommand()* method, depending on the command name that has been passed to it.

Example:

```
bool CHelloWorld::execHelloWorldCommand(const CCommandArgumentList &arguments)
{
      // Output a „hello world" message
      return true;
}

bool CHelloWorld::executeCommand(const std::string &command,
                                 const CCommandArgumentList &arguments)
{
      if ( command == "hello_world" )
      return execHelloWorld(arguments); else return false;
}
```

2.6 Step 6: Making the module accessible

After the application module class has been implemented, it has to be added to the framework's application module registry. There are two points within the framework, that have to be modified.

The first point is the file ***ApplicationModuleList.h***. This file contains the includes of header files, which contain the definitions of application module classes. To register a new application module, the header file of it's class has to be included by adding the proper *#include* statement to this file.

The second point is the file ***ApplicationModuleRegistry.h***. This file is the application module registry. It contains statements, that register the application modules to make them available to the framework. To register a new application module, a proper call to the *registerModule()* method of the *CApplication* class has to be added to this file. The instance of the application framework can be accessed through the variable *application* from inside this file.

Example:

```
// Add module to the application's module registry
application->registerModule(new ChelloWorldModule());
```

This statement adds the *HelloWorld* application module to the framework's module registry.

NOTE: The registered module instances are not destroyed by the framework automatically, i.e. they have to be destroyed by some other mechanism. Since typically there will be only one instance of an application module at the same time, it could be implemented using the ***singleton*** design pattern with a static smart instance pointer inside the class, to make the C runtime library take care of the module's destruction.

3 Interacting with the framework

All modules can use the functionality of the framework residing inside the *CApplication* class. The ***singleton*** design pattern was used for it's implementation to make sure, that there is only one instance of this class at the same time.

To access the framework's basic functionality, simply include the header file ***"CApplication.h"*** to your application module's source file. To obtain the instance pointer of the application, the static class method *getInstance()* has to be called.

Example:

```
#include „CApplication.h"
.
.
.
// Get a pointer to the application's instance
CApplication* app = CApplication::getInstance();
```

In this example the application's instance pointer is obtained and stored inside the *app* variable to make further access easier.

The interface of the *CApplication* class defines some methods to access the framework. How to use the most important ones will be described below. For more detailed information about the interface, take a look at the API documentation.

CApplication
CGlutMainWindow getMainWindow()*
CConsoleBuffer getConsoleBuffer()*
CObjectRepository getObjectRepository()*
int getModuleCount()
IApplicationModule getModule(int index)*
IApplicationModule getModule(const string &module_name)*
*void registerModule(IApplicationModule *module)*
*void unregisterModule(IApplicationModule *module)*
int getSubWindowCount()
CGlutSubWindow getSubWindow(int index)*
*void setActiveSubWindow(CGlutSubWindow *sub_window)*
CGlutSubWindow getActiveSubWindow()*
*void registerSubWindow(const std::string &name, CGlutSubWindow *sub_window)*
*void unregisterSubWindow(CGlutSubWindow *sub_window)*
int getCommandCount()
const CApplicationCommand getCommand(int index)*
const CApplicationCommand getCommand(const string &command)*
bool executeCommand(const string &command, const CCommandArgumentList &arguments)
*void initialize(int *argcp, char **argv)*
void run()
void terminate()
CApplication* getInstance()

*Table 2 - The public interface of the **CApplication** class*

3.1 Accessing the application's console

The basic part of the *MotionLab* framework's user interface is a console emulation based on *OpenGL*. It can be used to call the commands exposed by the registered application modules. Moreover, the application modules can use the console for the output of information or error messages.

The console can be accessed by calling the *getConsoleBuffer()* method of the *CApplication* class. This will return a pointer to the console's *CConsoleBuffer* object, which then can be used for text output. This object's interface is described in detail inside the API documentation.

3.1.1 Writing text to the console

Text can be written to the console either by calling the *print()* method of the *CConsoleBuffer* object or by using the standard C++ stream operators on it.

Example:

```
#include „CApplication.h"
// Get a pointer to the application's console
CConsoleBuffer* console = Capplication::getInstance()->getConsoleBuffer();

// Text output using print
console->print("Hello world!\n");

// Text output using stream operators
(*console) << "5 + 4 is:" << 5+4 << "\n";
```

3.1.2 Changing text and background color

The console provides colored output using 16 colors. The color in which the text will be printed can be changed using the *setTextColor()* and the *setBackgroundColor()* methods of the *CConsoleBuffer* object. The *enuConsoleColors* enumeration defines a list of color constants which can be used.

Example:

```
#include „CApplication.h"

// Get a pointer to the application's console
CConsoleBuffer* console = CApplication::getInstance()->getConsoleBuffer();

// Set text foreground and background color
console->setTextColor(cclBlack)
console->setBackgroundColor(cclWhite)

// Output text
console->print("This text is black on white!\n");
```

In this example white text on a black background is written to the console.

3.1.3 Clearing the contents of the console

The contents of the console can be cleared by calling the *clear()* method of the *CConsoleBuffer* object. The console will be cleared with the current background color.

3.2 Accessing the application's object repository

Application modules can exchange data using a central instance called the ***object repository*** to which objects of any type can be added and removed from. They can be accessed, by obtaining a pointer to their instance from the repository. It is possible as well, to lock an object against a certain access. The objects inside the repository are associated to names, meaning the repository behaves like a rudimentary file system.

A pointer to the object repository can be obtained by calling the *getObjectRepository()* method of the *CApplication* class, which will return a pointer to an *CObjectRepository* object. Detailed information about it's interface can be found inside the API documentation.

3.2.1 Adding objects to the repository

To add an arbitrary object to the object repository, it's instance first has to be wrapped into an *CObject* object, which then can be added to the object repository. The *CObject* class is a template class which simply contains a pointer to the instance of a dynamically allocated C++ object. Further information about the *CObject* class can be found inside the API documentation.

Example:

```
#include „CApplication.h“
#include „CObject.h“
#include „CSkeleton.h“

// Get the pointer to the object repository
CObjectRepository* rep = CApplication::getInstance()->getObjectRepository();

// Create a new skeleton
CSkeleton* skeleton = new CSkeleton();

// Create a hull for the skeleton
CObject<CSkeleton>* skeleton_object = new CObject<CSkeleton>(skeleton);

// Add the object to the repository
rep->addObject(“my_skeleton”,skeleton_object)
```

In this example, first an instance of the *CSkeleton* class is dynamically created. After that, a typed instance of the *CObject* class is created and the *CSkeleton* instance is assigned to it. In the last step, the object is added to the repository using the name *my_skeleton*. The chosen name has to be unique inside the object repository. From now on, the object can be accessed inside the object repository using this name.

NOTE: Objects which are stored inside an *CObject* object, must have been allocated dynamically using the *new()* operator. This is necessary, because when the surrounding *CObject* object is destroyed, the contained object instance is destroyed as well using the *delete()* operator.

3.2.2 Deleting objects from the repository

An object can be deleted from the repository using one of the versions of the *removeObject()* method of the *CObjectRepository* class. If the object is still in use, the method will return an error.

NOTE: The object will not only be removed from the repository, but it will be automatically destroyed as well.

Example:

```
#include „CApplication.h"

// Get the pointer to the object repository
CObjectRepository* rep = CApplication::getInstance()->getObjectRepository();

// Remove an object and destroy it
rep->removeObject("my_skeleton")
```

In this example, the object named *my_skeleton,* which was previously added to the repository, is removed from it (and thereby deleted).

3.2.3 Accessing objects inside the repository

It is possible, to obtain a pointer to an object stored inside the repository by calling one of the versions of the *CObjectRepository* class' *getObject()* method. This is the most complicated method of the object repository, because the desired access to the object has to be specified (read, write or read/write). Additionally, the object can be locked against a certain type of access, for example to prevent accidental modification by an instance, while the object is still in use by another instance. At last, the pointer to the caller's instance has to be provided to this method, since the object repository books the information about which callers have access to a certain object. A detailed description of this method can be found inside the API documentation.

Example:

```
#include „CApplication.h"
#include „CSkeleton.h"
.
.
.
// Get the pointer to the object repository
CObjectRepository* rep = CApplication::getInstance()->getObjectRepository();

// Get access to the object stored inside the repository
IObject* skeleton_object = NULL;
rep->getObject(this,"my_skeleton",afRead,lfWrite,skeleton_object);

// Get the instance stored inside the object
CSkeleton* skeleton = (CSkeleton*) skeleton_object->getInstance();
```

In this example, read access (access flag ***afRead***) to the object named *my_skeleton* is obtained and at the same tip it is locked for write access (lock flag ***lfWrite***).

3.2.4 Releasing objects

After an object, to which an instance pointer has been obtained using the *getObject()* method, has been used, it should be released. This can be achieved by calling one of the different *releaseObject()* methods defined by the *CObjectRepository* class. It will inform the object repository, that this object is no longer used by the caller and all access and lock information referred to the caller is removed from the repository. This does not influence the access of other callers to this object.

Example:

```
#include „CApplication.h"
#include „CSkeleton.h"
.
.
.
// Get the pointer to the object repository
CObjectRepository* rep = CApplication::getInstance()->getObjectRepository();

// Get access to the object stored inside the repository
IObject* skeleton_object = NULL;
rep->getObject(this,"my_skeleton",afRead,lfWrite,skeleton_object);

// Get the instance stored inside the object
CSkeleton* skeleton = (CSkeleton*) skeleton_object->getInstance();
.
. // Do something with the object
.
// Finally release the object
rep->releaseObject(this,"my_skeleton");
```

In this example, the object is released after it has been used.

3.2.5 Obtaining type information about an object inside the repository

It is possible to get information about the type of an object stored inside the object repository. For this, either the method *getObjectTypeName()* or the method *getObjectTypeInfo()* of the *CObjectRepository* class can be called. The first one returns the name of an object's type. The second one returns a pointer to an object's C++ *type_info* structure. This mechanism can be used to check an object's type before using it.

Example:

```
#include „CApplication.h"
#include „Cskeleton.h"

// Get the pointer to the object repository
CObjectRepository* rep = CApplication::getInstance()->getObjectRepository();

// Get an object's type information
const type_info* t_info;
rep->getObjectTypeInfo("my_skeleton",t_info);

// Check the type returned
if (t_info==typeid(CSkeleton))
{ // Do something }
else
{ // Return an error }
```

3.3 Sub windows

The user interface of the *MotionLab* application can be extended by adding sub windows to it. A sub window will be displayed inside the sub window panel located at the top half of the application's main window, the extents of the sub window will always be expanded to fill out the complete sub window panel.

For each registered sub window, a button will be added automatically by the framework to the application's window bar, so the user can switch between sub windows. These buttons can be used as well to close a sub window, which then will be automatically destroyed.

Additionally, the user is allowed to change the size of the sub window panel using a horizontal separator bar which can be moved vertically to set the size of the sub window panel and the console panel, meaning that the contents of the sub window should be made scalable or scrollable.

Sub windows are implemented by the *CGlutSubWindow* class using the GLUT library, as indicated by the name, but provide an object oriented interface. Since the sub window is a GLUT window, it also has an associated OpenGL render context. This means, that OpenGL commands can be used to perform drawing operations inside the window. The GLUT's callback mechanism has been replaced by a listener mechanism similar to the one implemented by the JAVA API. A detailed description of the classes *CGlutSubWindow*, *CGlutWindow, CGlutMainWindow* and the various window event listener interfaces can be found inside the API documentation.

3.3.1 Step 1: Creating a sub window

To create a sub window the instance of the application's main window must be retrieved first using the *getMainWindow()* method of the *CApplication* class. Then a new instance of the *CGlutSubWindow* class has to be created by calling it's constructor and specifying the desired OpenGL render context using some of the constants defined by the GLUT library. Additionally, a pointer to the sub window's parent window, which in this case must be the application's main window, and the sub window's extents have to be passed to the constructor.

Example:

```
#include „CGlutMainWindow.h“
#include „CGlutSubWindow.h“
#include „CApplication.h“

// Get the pointer to the main window instance
CGlutMainWindow* main_window = CApplication::getInstance()->getMainWindow();

// Create a new sub window with the main window as it's parent
CGlutSubWindow* sub_window = new CGlutSubWindow
                             (
                               GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH, // Render mode
                               main_window, // Parent window
                               0,0 // Window origin
                               640,480 // Window size
                             );
```

In this example, a new sub window is created with a double buffered OpenGL render context and assigned to the application's main window. The window's extents do not really matter, because, as mentioned above, the window will be stretched automatically to fill out the sub window panel it resides in.

3.3.2 Step 2: Registering the sub window

After the sub window has been created, it has to be added to the application framework's sub window registry. This can be done by calling the *registerSubWindow()* method of the *CApplication* class. After the window has been registered, the framework will take care of it. The window will be displayed inside the sub window panel and a button with the window's name will be added to the application's window bar.

Of course, a sub window can be unregistered by calling the *unregisterSubWindow()* method of the *CApplication* class.

Example:

```
#include „CGlutMainWindow.h“
#include „CGlutSubWindow.h“
#include „CApplication.h“

// Get the pointer to the main window instance
CGlutMainWindow* main_window = CApplication::getInstance()->getMainWindow();

// Create a new sub window with the main window as it's parent
CGlutSubWindow* sub_window = new CGlutSubWindow
                             (
                               GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH, // Render mode
                               main_window, // Parent window
                               0,0 // Window origin
                               640,480 // Window size
                             );

// Register the created sub window
CApplication::getInstance()->registerSubWindow(“mysubwindow”,sub_window);
```

In this example a new sub window is created with a double buffered OpenGL render context and assigned to the application's main window. After it's creation it is added to the application's sub window registry. The name displayed on the window button will be *mysubwindow*

NOTE: It is not necessary for the window name to be unique. If a window with the specified name does already exist, the framework will append a consecutive number to the new sub window's name.

3.3.3 Step 3: Adding window event listeners

Unless at least the listeners for the *reshape* and *display* events are attached to a window, nothing can be displayed inside the window, though making it useless.

A listener is simply a class implementing one (or more) of the interfaces defined by the abstract classes inside the ***WindowListeners.h*** header file. A listener for a particular window event can be added by calling one of the *addXListener()* (where *X* stands for the type of a window event) methods of the *CGlutWindow* class. A detailed description of the listener interfaces can be found inside the API documentation.

When a certain window event occurs, all listeners, which have registered for this event, will be notified by the window.

Example:

```
#include „CGlutMainWindow.h“
#include „CGlutSubWindow.h“
#include „CApplication.h“
#include „WindowListeners.h“

// Simple event listener class
class CMyWindowListener : public IDisplayListener
                          public IReshapeListener
{
      // Display event handler
      virtual void display(void* sender)
      {
            // Clear the frame buffer
            glClearColor(0,0,0,0);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            // Draw something using OpenGL commands
            ...
      }

      // Reshape event handler
      virtual void reshape(void* sender, int width, int height)
      {
            // Setup the window's viewport
            glViewport(0,0,width, height);

            // Setup projection and modelview matrix, etc.
            ...
      }

};

// Create an instance of the listener class
CMyWindowListener my_listener;

// Create and register sub window
.
.
.

// Attach listener to the created sub window
sub_window->addDisplayListener(&my_listener);
sub_window->addReshapeListener(&my_listener);
```

When the sub window is closed or destroyed, it can become necessary to destroy resources associated with it. This can be done by adding a listener for the window's destroy event by calling the *addDestroyListener()* method of the *CGlutWindow* class. The destroy event occurs, when the instance of the window is destroyed. The listener for this event could then free all additional resources, which have previously been associated with the window.

4 Extending the framework's core modules

Some advanced functionality of the framework is realized inside it's core application modules.

Module	*Description*
BasicCommands	This module implements some basic console commands. Those commands are: ***exit, cls, about, list, dump***
MassCalculator	This module is used for the calculation of a skeleton's mass properties. It implements the commands: ***calculate_mass, describe_algorithm***
FileLoader	This module makes it possible, to load data from files into the object repository. It implements only one command: ***load***
Visualization	This module implements visualization functionality. It can be used to visualize motion data streams attached to the nodes of a motion capture skeleton. It implements only one command: ***visualize***
Directory	This modules provides simple file system access. It implements the commands: ***cd***, ***dir***
ObjectRepository	This module gives the user control of the contents stored inside the object repository. It implements only one command: ***delete_object***

Table 3 - List of core application modules

Some of those modules provide points for extension. This section describes, how the extension mechanisms of each module work.

4.1 Extending the visualization module

The visualization module implements the ***visualize*** command. This command can be called with an arbitrary number of skeleton objects, which will be visualized by the module. Such a skeleton, which is implemented by the *CSkeleton* class, typically has data streams assigned to it's nodes. A data stream can store an arbitrary number of objects of the same type. A stream could for example contain the orientation matrices of a node for each frame of a motion capture sequence. Another possibility would be, to assign a stream of global positions to each node. These streams could be used by the visualization module to display the motion on the screen.

After the user has called the ***visualize*** command, a window is created and the motion capture scene is displayed inside the window. The user navigates through the scene using the mouse and controls the animation playback using the keyboard.

The visualization module can be extended by adding new visualization plug ins. This feature can be used to visualize new types of streams. One can imagine a plug in, that renders force streams, which have previously been calculated from a capture motion sequence, by displaying the force vectors. Since the visualization window has an associated OpenGL rendering context, the plug ins can use all available OpenGL commands to perform drawing operations.

4.1.1 Implementing a new visualization plug in

This section will explain, what has to be done to create a new visualization plug in and make it accessible to the framework. What this section will not explain, is how to implement a certain type of visualization.

A visualization plug in is simply a class implementing the *IGfxPlugin* interface.

IGfxPlugin
virtual void setEnabled (bool enabled) *virtual bool getEnabled () const* *virtual const char * getDescription () const* *virtual const char * getName () const* *virtual IGfxPlugin * create ()* *virtual void init (ISkeleton *skeleton, CGraphics *graphics)* *virtual void finalize (CGraphics *graphics)* *virtual void render (int frame, ISkeleton *skeleton, int skel_number)* *virtual void key (unsigned char key, int x, int y)*

*Table 4- The **IGfxPlugin** interface*

A detailed description of the *IGfxPlugin* interface can be found inside the API documentation.

4.1.1.1 Step 1: Creating a new visualization plug in class

The very first step is to create a class for the new visualization plug in, which has to derive from the *IGfxPlugin* interface.

Example:

```
#include "IGfxPlugin.h"

// A simple visualzation plugin
class CSimpleGfxPlugin : public IGfxPlugin
{
      // Here all methods defined by the IGfxPlugin interface
      // must be overwritten
};
```

4.1.1.2 Step 2: Exposing basic information about the plug in

Each plug in has to provide some basic information like it's name and a short description. This can be done by implementing the *getName()* and the *getDescription()* method.

Example:

```
// Return the plugin's name
const char* CSimpleGfxPlugin::getName()
{
      return "Simple";
}

// Return a short description of the plugin
const char* CSimpleGfxPlugin::getDescription()
{
      return "A simple visualization plugin";
}
```

4.1.1.3 Step 3: Plug in instance creation

To allow multiple visualization instances, a new set of plug in instances is created for each instance of the visualization. This requires a factory mechanism, which works as follows.

A basic set of plug in instances is added to the visualization module's plug in registry. Every time, the users invokes the ***visualize*** command, a new instance of the *CGraphics* class is created, which implements the visualization functionality. Then the *create()* method is called for each registered visualization plug in. The implementation of this method must simply create a new instance of the plug in class and return a pointer to it. This new instance is assigned to the *CGraphics* object created before, which allows each visualization to have it's private set of plug ins and makes it possible, to have private state data within a plug in.

Example:

```
IGfxPlugin* CSimpleGfxPlugin::create()
{
      // Simply create new instance of the plugin class
      return new CSimpleGfxPlugin();
}
```

4.1.1.4 Step 4: Plug in initialization and finalization

Before a scene is visualized, each registered plug in is initialized by the framework. To accomplish this, the framework calls the *init()* method on each registered plug in for each skeleton within the scene. The first parameter *skeleton* of this method is a pointer to the *ISkeleton* interface of the skeleton for which the plug in should be initialized. The plug in can use the methods of this interface to obtain pointers to the streams associated with each node of the skeleton.

The second parameter is a pointer to the parent *CGraphics* object. The *CGraphics* object implements the visualization core functionality. It creates the visualization window with an associated OpenGL rendering context, is responsible for the management of the skeletons to be visualized and for the management of the visualization plug ins, and it also implements camera handling, frame navigation and animation playback.

What has to be done exactly during initialization, depends on the type of the plug in. A plug in would typically check, if the streams needed for the visualization are really available. Or it could get the number of frames stored in a stream. Sometimes it could be necessary, to calculate a bounding box around the scene. Of course, other initialization tasks are possible too.

When the visualization is terminated, the framework calls the *finalize()* method for each registered plug in instance. The implementation of this method should release all resources that have been dynamically allocated during the call of the *init()* method. The parameter passed to this method is a pointer to the parent *CGraphics* object.

A detailed description of the *ISkeleton* interface and the *CGraphics* class can be found inside the API documentation.

4.1.1.5 Step 5: Implementing plug in activation / deactivation

A plug in can be enabled and disabled. Disabled plug ins are skipped during the rendering process. To enable or disable a plug in, the framework calls it's *setEnabled()* method. The implementation of this method should store the state passed to it through the *enabled* parameter. The implementation of the *getEnabled()* method should return the current state of the plug in, which was previously set by the *setEnabled()* method.

```
#include "IGfxPlugin.h"
class CSimpleGfxPlugin : public IGfxPlugin
{
protected:
	// Plugin state
	bool Enabled;

public:
	virtual void setEnabled(bool enabled)
	{
		// Store state
		Enabled = enabled;
	}

	virtual bool getEnabled()
	{
		// Return state
		return Enabled;
	}

};
```

4.1.1.6 Step 6: Implementing the drawing functionality

Every time a single frame has to be rendered, the visualization framework first clears the OpenGL frame buffer and depth buffer associated with the visualization window. Then the model/view matrix is initialized, since the visualization framework implements camera handling.

After these initialization tasks the *render()* method is invoked for each enabled plug in and for each skeleton which should be visualized. The first parameter *frame* is the number of the frame, which should be rendered. The second parameter *skeleton* is a pointer to the *ISkeleton* interface of the skeleton which should be rendered. The last parameter *skel_number* is the index number of the skeleton.

The implementation of this method can use all available OpenGL commands to perform drawing operations. All render state changes, which have been made to the rendering context, should be taken back after drawing has been finished, to ensure that the render state will not change between each plug in invocation. If the plug in needs to change the model view matrix, it should first preserve it by calling the *glPushMatrix()* function. After the drawing process has been finished, it should restore it by calling the *glPopMatrix()*. The same should be done if the projection matrix needs to be modified.

What the implementation should exactly do, depends on the type of the plug in. A plug in visualizing the positions of the skeleton's joints would simply obtain the positions of the skeleton's nodes from positional streams attached to the nodes and draw spheres. Another plug in may draw lines between the nodes. Many different implementations are possible.

4.1.1.7 Step 7: Adding user interaction

For user interaction, the plug in should implement the *key()* method. This method is invoked every time a key has been pressed inside the visualization window for each plug in. The first parameter of this method is the ASCII code of the key, which has been pressed. The other parameters store the current position of the mouse pointer relative to the visualization window's origin. The implementation of this method can be used, to give the user control of the plug in's rendering behavior.

To implement some advanced user interaction including mouse handling, an alternative method can be used. As mentioned above, the *init()* method of each plug in is called by the framework before visualization begins. One of the method's parameters is a pointer to the parent *CGraphics* object. By calling it's G*etGfxWindow()* method a pointer to the visualization window can be obtained. This pointer points to an instance of the *CGlutWindow* class. This class allows the registration of window event listeners which can be used to handle specific window events like keyboard or mouse input events. Adding listeners to a window is explained in section 3 of this document. More detailed information about the *CGlutWindow* class and the available listeners can be found inside the API documentation.

4.1.1.8 Step 8: Adding menu entries to the plug in sub menu

The parent *CGraphics* objects creates not only the visualization window, but also a pop up menu containing some basic commands. Additionally, a sub menu is created for each registered visualization plug in, containing one entry to disable or enable the plug in. The main menu and the plug in sub menu can be extended by a visualization plug in. To obtain a pointer to the visualization window's main menu, the the parent *CGraphics* object's G*etMainMenu()* method has to be called. It returns a pointer to a *CGlutMenu* object. To obtain a pointer to the sub menu of a plug in, the *GetPluginSubMenu()* of the parent *CGraphics* object has to be called, which also returns a pointer to a *CGlutMenu* object. The pointer to the instance of the plug in, whose sub menu should be returned, has to be passed to this function. The methods of the *CGlutMenu* object can be used to add new entries or change the existing ones. To handle menu entry clicks, a menu event listener has to be added to the sub menu. This can be done by calling the *addMenuListener()* method of the *CGlutMenu* class, which expects a pointer to the menu listener object. A menu event listener simply is a class implementing the *IMenuListener* interface. This interface defines only the *menu()* method, which should be implemented to handle the menu events. The first argument passed to this method is a pointer to the menu, whose entry has been clicked. The second one is a pointer to the menu entry object *CGlutMenuEntry,* which has been clicked by the user. A detailed description of the classes *CGlutMenu, CGlutMenuEntry* and *IMenuListener* can be found inside the API documentation.

The installation of new menu entries could be done inside the *init()* method of the plug in. Since this method can be called several times by the framework, the implementation must ensure, that the entries are added only once.

Example:

```
// Plug in initialization
void  CSimpleGfxPlugin::init(ISkeleton *skeleton, CGraphics *graphics)
{
      // Store pointer to the CGraphics object inside a member variable
      Graphics = graphics;

      // Make sure that the menu is initialized once
      if (!MenuInitialized)
      {
            // Get a pointer to the sub menu of the plug in
            CGlutMenu* sub_menu = graphics->GetPluginSubMenu(this);
            // Add a new menu entry
            sub_menu->addEntry("Hello World!");
            // Add the plug in as a menu listener to the sub menu
            sub_menu->addMenuListener(this);
            MenuInitialized = true;
      }
}
// Menu event handler
void CSimpleGfxPlugin::menu(void *sender, CGlutMenuEntry *menu_entry)
{
      // Check if the sender is the plug in sub menu
      if (sender == Graphics->getPluginSubMenu(this))
      {
            // Check which entry has been clicked and do something
      }
}
```

In the example above, the *init()* method is used to add a new entry to the plug in's sub menu. To ensure that this is done only once, the *MenuInitialized* member variable has been used. The implementation of the menu event handler first checks if the menu which raised the event is the plug in sub menu. Therefore the fictive member variable *Graphics* is used, which has been initialized with a pointer to the instance of the parent *CGraphics* object inside the *init()* method. The *sender* parameter then is compared to the result of the *getPluginSubMenu()* method, called with the instance pointer of the current plug in passed to it.

NOTE: If the plug in is meant to be added as a menu event handler to the sub menu, it has to derive from the *IMenuListener* class.

4.2 Extending the file loader module

The file loader module implements the ***load*** command, which can be used to load data from files into an object and then store this object inside the object repository. Typing ***"load s1 skeleton.asf motion.amc"***at the console prompt for example loads first the skeleton file **skeleton.asf** into a skeleton object, loads then the motion capture data from ***motion.amc*** into the skeleton object and finally stores the skeleton object inside the object repository using the name ***s1***.

The file loader module can also be extended by registering so called ***file import filters*** and thereby adding support for new file formats.

4.2.1 Implementing a file import filter

A file import filter is a class implementing the *IFileImportFilter* interface.

IFileImportFilter
void initialize() *void finalize()* *void setErrorHandler(IErrorHandler *error_handler)* *IErrorHandler * getErrorHandler()=0* *void setObjectRepository(CObjectRepository *object_repository)* *CObjectRepository * getObjectRepository()* *const std::string & getName ()* *const std::string & getDescription()* *int getExtensionCount()* *const std::string & getExtension(int index)* *bool loadFiles (const std::string &destination_object, const std::vector< std::string > &files)*

Table 5- The **IFileImportFilter** interface

A detailed description of the *IFileImportFilter* interface can be found inside the API documentation.

This section will lead you step by step through the process of implementing a new file import filter and making it accessible by the framework. What this section does not show, is how to load a certain file format.

4.2.1.1 Step 1: Creating a new file import filter class

The first step is to create a class for the new file import filter. This class has to derive from the *IFileImportInterface*.

Example:

```
#include "IFileImportFilter.h"

// A simple file import filter class
class CSimpleImportFilter : public IFileImportFilter
{
      // Here all methods defined by the IFileImportFilter interface
      // must be overwritten
};
```

The next step would be to implement all methods defined by the interface.

4.2.1.2 Step 2: Exposing basic information about the file import filter

Each file import filter has to expose it's name and a short description. This is done by implementing the methods *getName()* and *getDescription()*. The first returns a reference to a string object containing the filter's name, the latter returns a reference to a string object containing the filter's description.

NOTE: The name of the module must be unique.

This information will be displayed for each file import filter when the user types ***"list import_filters"*** at the console prompt.

Example:

```
// Returns the name of the filter which must be unique
const std::string& CSimpleImportFilter::getName()
{
      static std::string Name = "SimpleImportFilter";
      return Name
}

// Returns a short description of the filter
const std::string& CSimpleImportFilter::getDescription()
{
      static std::string Description = "A simple file import filter"
      return Description
}
```

4.2.1.3 Step 3: Object repository handling

Before the file import filter can load something, it must be told by the framework, to which object repository the loaded data should be written. To accomplish this, the framework calls the *setObjectRepository()* method of the file import filter class. The implementation of this method should store the pointer to the object repository, passed through the *object_repository* parameter for further use. The implementation of the *getObjectRepository()* method should return the pointer to the object repository which has been set using the *setObjectRepository()* method.

Example:

```
#include "IFileImportFilter.h"
#include "CObjectRepository.h"

// A simple file import filter class
class CSimpleImportFilter : public IFileImportFilter
{
protected:

      // Pointer to the object repository to which the loaded
      // files should be written
      CObjectRepository* ObjectRepository;

public:

      // Sets the destination object repository
      virtual void setObjectRepository(CObjectRepository* object_repository)
      {
            // Store pointer
            ObjectRepository = object_repository;
      }

      // Return the destination object repository
      virtual CobjectRepository* getObjectRepository()
      {
            // Return pointer
            return ObjectRepository;
      }

};
```

More information about the use of the object repository can be found inside the API documentation or in section 3 of this document.

4.2.1.4 Step 4: Error handling

An important part in error handling is the notification of the user, i.e. if a file is corrupt or in an incorrect format, the user should be informed about what went wrong. To make this possible, the framework associates an error handler with each file import filter module. This error handler implements the *IErrorHandler* interface, whose description can be found inside the API documentation. Here it's implementation writes the error message to the application's console, so this object can be used for error output.

The framework calls the *setErrorHandler()* method for each registered file import filter to set it's error handler. The implementation of this method should store the instance pointer passed through the *error_handler* inside the filter class for further use. The *getErrorHandler()* method should return the pointer to the error handler previously set by the *setErrorHandler()* method.

A typical implementation of both methods would be similar to the implementation shown in the example above for the methods *setObjectRepository()* and *getObjectRepository()*.

4.2.1.5 Step 5: Filter initialization and finalization

Each import filter will be initialized by the framework at application startup and finalized at application shutdown. To accomplish this, the framework calls the *initialize()* method of each registered file input filter when the application starts and *finalize()* when the application is terminated, meaning if the file import filter needs to initialize dynamic resources at startup and release them later, both methods should be implemented. Otherwise their implementation should be left empty.

4.2.1.6 Step 6: Exposing information about supported file formats

Each file import filter has to provide information to the framework about the supported file formats. A filter may support more than one file format at the same time and must return a set of file extensions of the file formats it supports. Therefore, the methods *getExtensionCount()* and *getExtension()* have to be implemented. The method *getExtensionCount()* returns the number of file extensions supported by the filter or zero, if it does not support any file format. The method *getExtension()* returns the name of the supported extension designated by the *index* parameter, with the first extension having an index of zero and the last one an index of *getExtensionCount()-1*. If the index is out of range, the method should return an empty string (“”). The extension strings must include the trailing dot (i.e ***.txt*** instead of ***txt***), and the sets of supported extensions must not overlap, meaning that it is not allowed, to have two different file import filters supporting the same file type.

Example:

```
// Returns the number of supported file extensions
const std::string& CSimpleImportFilter::getExtensionCount()
{
      // The filter supports one extension
      return 1;
}

// Returns the specified supported file extension
const std::string& CsimpleImportFilter::getExtension(int index)
{
      static std::string TextExtension = ".txt";
      static std::string NULL_STR = "";
      // Check index and return specified extension
      if (index==0) return TextExtension; else return NULL_STR;
}
```

4.2.1.7 Step 7: Loading files

The loading process has to be implemented inside the *loadFiles()* method defined by the *IFileImportFilter* interface. When the user enters the ***"load"*** command at the console, it's first parameter has to be the name, under which the loaded data should be stored inside the object repository. This name is passed as the *destination_object* parameter to the *loadFiles()* method. The name of the destination object has to be followed by one ore more filenames, which should be loaded into the object and though making it possible to load data, which is split up into several files, into one object. This list of filenames is passed as the second parameter *files* to the *loadFiles()* method. It's type is *const std::vector< std::string >&*, so the number of files passed can be obtained by calling the method *size()* of the C++ vector class. The filenames can be accessed using the array operator, the first file having an index of zero. Since the user can specify a mixture of different file types in the ***load*** command, the framework checks the extensions of the specified files and binds them into groups, so that each group contains only files which can be loaded by a single import filter. This means, that the files are dispatched only to a filter, which is capable of loading them.

The implementation of the *loadFiles()* method should perform the following steps:

- Check, if the specified files do really exist. If a file cannot be found, ***false*** should be returned.
- Check, if the object specified by the *destination_object* parameter does already exist inside the object repository. If not, a new object must be created. Otherwise the following implementations are possible:
  1. An error message could be written to the console using the associated error handler. The method should return ***false*** then.
  2. The data stored inside the object could be overwritten.
  3. The new data could be appended to the existing object.
- Load the data from the specified files into an object. For example, the ASF/AMC file import filter creates an instance of the *CSkeleton* class, loads the skeleton structure data from the *ASF* file and finally loads the motion capture data from the *AMC* file into the created skeleton.
- Create a *CObject* wrapper for the object, containing the data loaded from the files. A detailed documentation of the *CObject* template can be found inside the API documentation. This wrapper is needed, because the object repository is only capable of storing *IObject* instances or it's derivatives.
- Store the *CObject* wrapper inside the object repository under the name specified by the *destination_object* parameter. Section 3 of this document explains, how the object repository can be used. A detailed information about the *CObjectRepository* class can also be found inside the API documentation.

After all steps have been executed successfully, the method should return ***true.*** Otherwise an error message should be displayed using the associated *IErrorHandler* object and ***false*** should be returned, signalizing that the loading process was not successful.

4.2.1.8 Step 8: Making the file import filter accessible

After the file import filter class has been implemented, it has to be added to the framework's file filter registry. There are two points within the framework, which have to be modified.

The first point is the file ***FileImportFilterList.h***. It contains the includes of header files which contain the definitions of file import filter classes. To register a new file import filter, the header file of it's class has to be included by adding the proper *#include* statement to this file.

The second point is the file ***FileImportFilterRegistry.h***. It is the file import filter registry and contains statements, which register the file import filters to make them available to the framework. To register a new file import filter, a proper call to the *registerImportFilter()* method of the *CFileLoaderModule* class has to be added to this file. Since the registry file is directly included into the implementation of this class, the *registerImportFilter()* method can be called directly.

Example:

```
// Add file import filter to the registry
registerFilter(new CSimpleImportFilter());
```

This statement would add the simple import filter to the framework's import filter registry.

NOTE: The registered filter instances are not destroyed by the framework automatically, so they have to be destroyed by some other mechanism. Since there will typically only be one instance of a file import filter at the same time, it could be implemented using the ***singleton*** design pattern with a static smart instance pointer inside the class, to make the C runtime library taking care of the filter's destruction.

4.3 Extending the mass calculation module

For more advanced analysis processes, as for example for physically based motion analysis, mass properties of a motion capture skeleton have to be calculated. The implementation of this functionality resides inside the mass calculation module, which allows a specialized mass calculation algorithm to be executed. It calculates the mass properties of a skeleton model, which is defined by the algorithm itself. After this process a customizable script is executed to perform an assignment of the mass properties calculated by the algorithm to the bones of a motion capture skeleton. This is necessary, because the structure of the skeleton model the algorithm is based on, could differ from the structure of the motion capture skeleton.

The module can be extended in two ways. One way is to add new mass calculation algorithms. The other way is to extend the scripting language by adding new operations. Both extension mechanisms are explained below.

4.3.1 Adding a new mass calculation algorithm

This section explains, what has to be done to implement a new mass calculation algorithm and to make it accessible by the framework. What this section does not show, is how to implement a specific kind of calculation algorithm.

A mass calculation algorithm is a class implementing the *IMassCalculationAlgorithm* interface.

IMassCalculationAlgorithm
*const char * getName()* *int getSegmentCount()* *const char * getSegmentName(int index)* *const char * getSegmentDescription(int index)* *unsigned int getSegmentFields(int index)* *int getParameterCount()* *const char * getParameterName(int index)* *const char * getParameterDescription(int index)* *bool calculate(CSegmentList *segment_list, int param_count, real params[])*

Table 6 - The IMassCalculationAlgorithm interface

A detailed description of the IMassCalculationAlgorithm interface can be found inside the API documentation.

4.3.1.1 Step 1: Creating a new mass calculation algorithm class

The first step is to create a class for the new file import filter. This class must derive from the *IFileImportInterface*.

Example:

```
#include "IMassCalculationAlgorithm.h"

// A simple mass calculation algorithm
class CSimpleMassAlgorithm : public IMassCalculationAlgorithm
{
      // Here all methods defined by the IMassCalculationAlgorithm interface
      // have to be overwritten
};
```

4.3.1.2 Step 2: Exposing basic information about the algorithm

Each algorithm must have name, which should be unique and returned by the implementation of the *getName()* method.

Example:

```
// Return the name of the algorithm
const char* CSimpleMassAlgorithm::getName()
{
      static const char* Name = ”Simple”;
      return Name;
};
```

4.3.1.3 Step 3: Exposing information about the skeleton structure

Each algorithm has to expose information about the skeleton structure it's calculations are based on. A skeleton is assumed to consist of a set of segments, for which a set of mass properties is calculated by the algorithm. A simple algorithm for example might assume the human hand to be only one segment. On the other hand a more advanced algorithm might additionally calculate mass properties for each finger of the hand.

To expose this information, the algorithm has to implement the methods *getSegmentCount*(), *getSegmentName()*, *getSegmentDescription()* and *getSegmentFields()*. The implementation of the *getSegmentCount()* returns the number of segments the skeleton model consists of. The implementation of the *getSegmentName()* method returns the name of the segment designated by the *index* parameter. If the *index* parameter is out of range, a ***NULL*** pointer should be returned. The segment names should be unique within the algorithm. The implementation of the *getSegmentDescription()* returns a short description of the segment designated by the *index* parameter. If the *index* parameter is out of range, a ***NULL*** pointer should be returned. The *getSegmentFields()* returns a bit mask for the skeleton segment designated by the *index* parameter. This bit mask should be a combination of the constants defined by the *enuSegmentFields* enumeration. It gives information, about which mass properties will be calculated by the algorithm for the specified segment. Possible properties are the mass, the center of mass, the moment of inertia and the segment's length. If the *index* parameter is out of range, the method should return zero.

The information provided by these methods will be displayed, when the user calls the ***describe_algorithm*** command on a mass calculation algorithm.

Example:

```
// Return number of supported segments
int CSimpleMassAlgorithm::getSegmentCount()
{
  return 1;
}

// Return the name of a segment
const char* CSimpleMassAlgorithm::getSegmentName(int index)
{
      if (index==0) return ”LowerArm”; else return NULL;
}
```

```
// Return the description of a segment
const char* CSimpleMassAlgorithm::getSegmentDescription(int index)
{
      if (index==0) return ”The lower arm of a human”; else return NULL;

}
// Return the mass properties calculated for a segment
unsigned int CSimpleMassAlgorithm::getSegmentFields(int index)
{
      // Only the center of mass and the moment of inertia are calculated
      if (index==0) return flMass | flMomentOfInertia ; else return 0;
}
```

In this example, the skeleton consists only of one segment called ***LowerArm*** for which only the mass and the moment of inertia are calculated.

4.3.1.4 Step 4: Exposing information about supported parameters

A mass calculation algorithm can be made parameterizable, the only limitation being, that all parameters have to be numeric. To support parameters, the methods *getParameterCount()*, *getParameterName()* and *getParameterDescription()* have to be implemented. The first method returns the number of parameters expected by the calculation algorithm. If no parameters are required for the algorithm, this method should return zero. The information provided by this function is needed by the ***calculate_mass*** command to check, whether enough parameters have been specified by the user. The implementation of the *getParameterName()* method returns the name of the parameter designated by *index*. The *getParameterDescription()* method returns a short description of the parameter designated by *index*. The information provided by these methods will be displayed, when the ***describe_algorithm*** command is executed by the user on a mass calculation algorithm.

Example:

```
// Return number of supported parameters
int CSimpleMassAlgorithm::getParameterCount()
{
  return 2;
}
// Return the name of a parameter
const char* CSimpleMassAlgorithm::getParameterName(int index)
{
      if (index==0) return ”mass”;
      else
      if (index==1) return ”height”;
      else
      return NULL;
}
// Return the description of a parameter
const char* CSimpleMassAlgorithm::getParameterDescription(int index)
{
      if (index==0) return ”Total body mass”;
      else
      if (index==0) return ”Total body height”;
      else
      return NULL;
}
```

In this example, the algorithm supports two parameters (total body mass and total body height) on which it's calculation will be based.

4.3.1.5 Step 5: Calculating the mass properties

When the user executes the ***calculate_mass*** command, an algorithm and it's parameters can be specified. The framework parses the parameters and calls the *calculate()* method of the specified algorithm. This method should perform the calculation of the mass properties. The first parameter passed to the method is a pointer to a *CSegmentList* object. A detailed description of this class can be found inside the API documentation. This object is used to store the calculated results. It should be cleared at the beginning using the *clear()* method. The second parameter is the number of parameters passed to the algorithm, and the last parameter is an array containing the values of these parameters. For each segment which is supported by the algorithm, the following has to be done:

- Create a *CSegment* object. A detailed description of the *CSegment* class can be found inside the API documentation. The name of the new object should be the same as the name of the segment.
- Calculate the supported mass properties for the segment and copy the computed values into the *CSegment* object created before.
- Add the created *CSegment* object to the segment list passed to the algorithm by the *segment_list* parameter.

If the properties could be computed successfully, the method should return ***true*****,** otherwise ***false***.

4.3.1.6 Step 6: Making the new algorithm accessible

After the mass calculation algorithm class has been implemented, it has to be added to the framework's algorithm registry. There are two points within the framework, which have to be modified.

The first point is the file ***MassCalculationAlgorithmList.h***. It contains the includes of header files, which contain the definitions of mass calculation algorithm classes. To register a new mass calculation algorithm, the header file of it's class has to be included by adding the proper *#include* statement to this file.

The second point is the file ***MassCalculationAlgorithmRegistry.h***, which is the mass calculation algorithm registry. It contains statements, which register the mass calculation algorithms to make them available to the framework. To register a new mass calculation algorithm, a proper call to the *registerAlgorithm()* method of the *CMassCalculator* class has to be added to this file. The instance of the mass calculator class can be accessed through the *mass_calculator* variable from inside this file.

Example:

```
// Add mass calculation algorithm to the registry
mass_calculator->registerAlgorithm(new CsimpleMassAlgoritm());
```

This statement adds the simple mass algorithm to the framework's algorithm registry.

NOTE: The registered algorithm instances are not destroyed by the framework automatically, , so they have to be destroyed by some other mechanism. Since there will typically only be one instance of an algorithm at the same time, it could be implemented using the ***singleton*** design pattern with a static smart instance pointer inside the class, to make the C runtime library taking care of the filter's destruction.

4.3.2 Adding a new script operation

As mentioned above, the mass calculation module has an integrated script interpreter. It is very simple and it's syntax is comparable to a typical assembly language. Each line contains only one operation followed by it's operands. A line can be a comment starting with the *"#"* character as well. Additionally, labels can be defined to support jumps and calls. In this case a line contains only one identifier starting with the *":"* character. The script is executed sequentially (line by line). The operations of the script can modify the contents of an operand stack and the contents of a symbol table storing values associated to names.

Although the syntax of the script interpreter cannot be changed, it can be extended by adding new operations to it. A script operation simply is a class implementing the *IOperation* interface.

IOperation
const string & getName() *bool checkArguments(const CArgumentList &arguments, CLabelList &label_list,* *unsigned int instruction_line, IErrorObject &error_object)* *bool execute(const CArgumentList &arguments, CContext &context, IErrorObject &error_object)*

Table 7 - The IOperation interface

This section describes, what has to be done to implement a new operation and add it to the script interpreter.

4.3.2.1 Step 1: Creating a new operation class

The first step is to create a class for the new operation. This class must derive from the *IOperation* interface.

Example:

```
#include "IOperation.h"

// A simple operation
class CSimpleOperation : public IOperation
{
      // Here all methods defined by the Ioperation interface
      // have to be overwritten
};
```

4.3.2.2 Step 2: Exposing the name of the operation

Each operation has a name, which has to be unique, since operation overloading is not supported by the script interpreter. The name of the operation is also used to call it from inside the script. To expose the name, the *getName()* method has to be implemented, which should return a constant reference to a string containing the name of the operation.

Example:

```
// Return the name of the operation
const std::string& CSimpleOperation::getName()
{
      static std::string Name = "simple"; return Name;
};
```

4.3.2.3 Step 3: Checking immediate arguments

The call of an operation can be followed by immediate arguments, which are passed directly to the operation. Since these arguments are available instantly, they can be checked before the script is executed. This is done in the very first pass. After a line has been scanned, the script interpreter determines which operation has to be called in this line. Then the *checkArguments()* method of the operation is called and the immediate arguments are passed to it. It must validate these arguments and return ***true*** if the arguments are valid, ***false*** otherwise. What should be checked is, whether the number of parameters which have been passed to the operation is valid and if the types of these parameters are valid too.

The first parameter *arguments* of the *checkArguments()* method is a *CArgumentList* object, containing a list of immediate arguments which were passed to the operation. Each entry in this list is a *CArgument* object, whose methods can be used to obtain information about the type of the argument and it's value.

The second parameter *label_list* is a list of labels defined inside the script. This list is needed for validation of a jump or call operation's destination.

The third parameter *instruction_line* is the number of the line containing the call of the operation.

The last parameter *error_object* is a reference to an instance of the *IErrorObject* class. This instance can be used to output error messages. The *enuScriptInterpreterErrors* enumeration defines constants for the most common errors which can occur during script execution. They should be used to identify an error when calling the *raiseError()* method of the error object passed to the *checkArguments()* method. A detailed description of the *IErrorObject* class and the *enuScriptInterpreterErrors* enumeration can be found inside the API documentation.

Example:

```
bool CSimpleOperation::checkArguments(const CArgumentList &arguments,
                                      CLabelList &label_list,
                                      unsigned int instruction_line,
                                      IErrorObject &error_object)
{
      // Check number of arguments
      if (arguments.count()!=1)
      {
            // Raise error, since exactly one argument is allowed
            error_object.raiseError(this,ERROR_INVALID_ARGUMENT_COUNT,
                                    "Exactly one argument accepted!");
            return false;
      }

      // Check if argument is numeric
      if (arguments[0]->getType()!= atNumeric)
      {
            // Raise error, since type is invalid
            error_object.raiseError(this,ERROR_INVALID_ARGUMENT_TYPE,
                                    "NUMERIC argument expected!");
            return false;
      }

      // All checks passed
      return true;
}
```

4.3.2.4 Step 4: Implementing the behavior of the operation

The behavior of the operation must be implemented inside the *execute()* method, which is called by the script interpreter every time the operation should be executed. It must return ***true,*** if the operation could be executed successfully and ***false*** otherwise.

The first parameter *arguments* of the *execute()* method is a *CArgumentList* object containing a list of immediate arguments, which were passed to the operation. Each entry in this list is a *CArgument* object, whose methods can be used to obtain information about the type of the argument and it's value.

The second parameter *context* is a reference to a *CContext* object containing information about the current execution context of the script interpreter. The context contains the operand stack, the symbol table, the label list, the current instruction pointer and the number of the line which is executed. By using the methods of the *CContext* object, the instance pointer to the objects listed before can be obtained.

The last parameter *error_object* is a reference to an instance of the *IErrorObject* class. It can be used to output error messages. The *enuScriptInterpreterErrors* enumeration defines constants for the most common errors, that can occur during script execution. They should be used to identify an error when calling the *raiseError()* method of the error object passed to the *execute()* method. A detailed description of the *IErrorObject* class and the *enuScriptInterpreterErrors* enumeration can be found inside the API documentation.

Typically, an implementation of the *execute()* method would at first call the *checkArguments()* method to check, if the immediate arguments are valid. After the arguments have been checked, the operation is allowed to modify the context of the script interpreter. It could for example modify the contents of the operand stack or the symbol table, but is allowed as well to modify the instruction pointer, so that control of the program flow can be implemented.

The operand stack can be accessed by calling the *getOperandStack()* method of the *CContext* class, which returns a pointer to a *COperandStack* object. Instances of the *COperand* class can be pushed onto the stack or removed from it. The elements of the stack can also be accessed directly by calling the *getOperand()* method of the *COperandStack* and specifying an element index, zero being the topmost operand. The operation should always check, if the stack contains enough operands before using them. If not, the error object should be used to raise an ERROR_STACK_UNDERFLOW error and the operation should be terminated returning ***false***.

The symbol table can be accessed by calling the *getSymbolTable()* method of the *CContext* class, which returns a pointer to a *CSymbolTable* object. The symbol table is a list of names, to which *CSymbol* instances are assigned. It can be used to implement variables. New name/*CSymbol* pairs can be added to it. Existing ones can be removed or accessed calling the *find()* method and specifying the name of the symbol which should be returned. If a symbol cannot be found, the operation should raise an ERROR_SYMBOL_NOT_FOUND error and return ***false***.

The label list can be accessed by calling the *getLabelList()* method of the *CContext*, which returns a pointer to a *CLabelList* object. The label list contains a list of names associated with instruction pointer values. These can be used to implement jump and call operations. To change the position of the instruction pointer, the *setInstructionPointer()* method can be used with one of the values contained inside the label list.

More detailed information about the classes *CContext*, *COperandStack, CsymbolTable, CLabelList, COperand, CSymbol*, *IObject* and *CObject* can be found inside the API documentation.

Example:

```
bool CSimpleOperation::execute(const CArgumentList &arguments,
                                    CContext &context,
                                    IErrorObject &error_object)
{
      // Check immediate arguments
      if (checkArguments(arguments,*context.getLabelList(),
            context.getInstructionLine(),error_object))
      {

            // Get operand stack
            COperandStack* operand_stack = context->getOperandStack();

            // Create a string object and copy the value of the
            // first argument to it
            CStringObject* string_object = NULL;
            string_object = new CStringObject(arguments[0]->getValue());

            // Push the string object onto the operand stack
            operand_stack->push(new COperand(string_object))

            // Execution was succesful
            return true;
      }

      // Execution failed
      return true;

}
```

In this example, the immediate arguments are checked first by calling the operation's *checkArguments()* method. A part of the required parameters, like the label list and the current instruction line, are taken from the context object. Then a pointer to the operand stack is obtained by calling the *getOperandStack()* method of the context object. In the next step, a string object is created and the value of the first immediate argument is assigned to it. In the last step, the string object is pushed as an operand onto the operand stack by creating a new *COperand* object and assigning the string object to it. Then ***true*** is returned to signalize, that the operation has been executed successfully.

4.3.2.5 Step 5: Making the operation accessible

After the operation class has been implemented, it has to be added to the framework's script operation registry. There are two points within the framework which have to be modified.

The first point is the file ***ScriptOperationList.h***. It contains the includes of header files, which contain the definitions of script operation classes. To register a new script operation, the header file of it's class has to be included by adding the proper *#include* statement to this file.

The second point is the file ***ScriptOperationRegistry.h***. It is the script operation registry and contains statements, which register the script operations to make them available to the script interpreter. To register a new script operation, a proper call to the *registerOperation()* method of the *CScriptInterpreter* class has to be added to this file. The instance of the script interpreter class can be accessed through the *ScriptInterpreter* variable from inside this file.

Example:

```
// Add mass calculation algorithm to the registry
ScriptInterpreter.registerOperation(new CSimpleOperation());
```

This statement adds the simple operation to the script interpreter's operation registry.

NOTE: The registered operation instances are destroyed by the script interpreter automatically when the application terminates, so it is not necessary to free the instances explicitly.

4.4 Extending the basic commands module

As stated above, the basic commands module implements the most basic console commands. The ***exit*** command terminates the application, the ***cls*** command clears the contents of the console, the ***about*** command writes information about the application and it's authors to the console. The commands ***dump*** and ***list*** are most interesting, because their functionality can be extended by registering special handlers.

4.4.1 Extending the list command

The ***list*** command is typically used to output a list of objects of a certain type. It can for example be used to output a list of available commands by typing ***"list commands"*** or to output a list of objects contained inside the object repository by typing ***"list objects"***.

The command can be extended by adding new types of objects which can be listed. This is done by registering so-called list handlers which will be responsible for the output of a list of these objects.

4.4.1.1 Step 1: Implementing the IListObjectHandler interface

A list handler is a class implementing the interface defined by the abstract class ***IListObjectHandler:***

IListObjectHandler
void listObjects(const std::string &object_type)

*Table 8- The **IListObjectHandler** interface*

The ***IListObjectHandler*** interface only defines the *listObjects()* method. The implementation of this method should output a list of objects of the type specified by the *object_type* parameter to the application's console.

Example:

```
#include "IListObjectHandler.h"

// Simple list handler class
class CListCommandsHandler : public IListObjectHandler
{
      void listObjects(const std::string &object_type)
      {
            // Check if the object type is „commands"
            if (object_type=="commands")
            {
                  // Output a list of available commands to the console
            }
      }
};
```

This example shows, what a simple list handler class could look like. The *CListCommandsHandler* class derives from the *IListObjectHandler* class and implements it's *listObjects()* method. The implementation checks if the *object_type* parameter is equal to *"commands"*. In that case, a list of the available commands is written to the console. Otherwise the handler does nothing.

A handler can be responsible for listing several object types at the same time. To allow this, the *object_type* parameter is provided, so the handler can use it to examine, which type of objects should be listed.

4.4.1.2 Step 2: Registering the handler

After the handler has been implemented, it has to be added to the basic commands module's registry of list handlers. Therefore a pointer to the instance of the basic commands module has to be obtained first. This can be done by including the ***"CBasicCommandsModule.h"*** header file into the handler's source file and calling the static method *getInstance()* of the *CBasicCommandsModule* class.

Example:

```
#include "IListObjectHandler.h"
#include "CBasicCommandsModule.h"
.
. // List handler class definition goes here
.
// Get the pointer to the instance of the basic commands module
CBasicCommandsModule* module = CBasicCommandsModule::getInstance();
```

After the pointer has been obtained, the handler can be registered by calling the *registerListHandler* method defined inside the *CBasicCommandsModule* class. The first parameter of this method is the name of the object's type, for which the handler is to be registered. The second one is a pointer to the handler's instance, which should be registered. Detailed information about the *registerHandler()* method can be found inside the API documentation.

Example:

```
#include "IListObjectHandler.h"
#include "CbasicCommandsModule.h"
// Simple list handler class
class CListCommandsHandler : public IListObjectHandler
{
      void listObjects(const std::string &object_type)
      {
            // Implementation of handler goes here
      }
};
.
.
.
// Get the pointer to the instance of the basic commands module
CBasicCommandsModule* module = CBasicCommandsModule::getInstance();

// Create an instance of the CListCOmmandsHandler class
CListCommandsHandler ListCommandsHandler

// Register handler for the object type "commands"
module->registerListHandler("commands",&ListCommandsHandler);
```

This example defines a handler class, creates an instance of the class and registers this instance as a list handler. From now on, every time the user types ***"list commands"***, the *listObjects()* method of the registered handler will be called with the value **"commands"** passed to it.

Of course, a handler can also be unregistered by calling the *unregisterListHandler()* method of the *CBasicCommandsModule* class and passing the type of object for which the handler should be removed.

NOTE: The instances of registered handlers are not destroyed automatically by the framework. This must be done explicitly.

4.4.2 Extending the dump command

The ***dump*** command is used to output the contents of an object stored inside the object repository. It can for example be used to output the structure of a motion capture skeleton to the console by typing ***"dump s1"***, *s1* being a skeleton object contained inside the object repository.

This command can be extended by adding a so-called ***dump handler*** for each type of object whose contents can be dumped.

4.4.2.1 Step 1: Implementing the IDumpObjectHandler interface

A dump handler is a class implementing the interface defined by the abstract class ***IDumpObjectHandler:***

IDumpObjectHandler
void dumpObject(IObject object)*

*Table 9- The **IDumpObjectHandler** interface*

The ***IDumpObjectHandler*** interface only defines the *dumpObject()* method. The implementation of this method should output the contents of the object referenced by the *object* parameter to the application's console.

Example:

```
#include "CSkeleton.h"
#include "IDumpObjectHandler.h"

// Simple dump handler class
class CDumpSkeletonHandler : public IDumpObjectHandler
{
      void dumpObject(IObject* object)
      {
            // Check if the object's type is "CSkeleton"
            if (object->getTypeInfo()==typeid(CSkeleton))
            {
                  // Output the structure of the skeleton to the console
            }
      }
};
```

This example shows, what a simple dump handler class could look like. The *CDumpSkeletonHandler* class derives from the *IDumpObjectHandler* class and implements it's *dumpObject()* method. The implementation checks, if the type of the object passed through the *object* parameter is *CSkeleton*. In that case, the structure of the skeleton is written to the console. Otherwise the handler does nothing.

4.4.2.2 Step 2: Registering the handler

After the handler has been implemented, it has to be added to the basic commands module's registry of dump handlers. For this, a pointer to the instance of the basic commands module must be obtained first. This can be done by including the ***"CBasicCommandsModule.h"*** header file into the handler's source file and calling the static method *getInstance()* of the *CBasicCommandsModule* class.

Example:

```
#include "IDumpObjectHandler.h"
#include "CBasicCommandsModule.h"
.
. // Dump handler class definition goes here
.
// Get the pointer to the instance of the basic commands module
CBasicCommandsModule* module = CBasicCommandsModule::getInstance();
```

After the pointer has been obtained, the handler can be registered by calling the *registerDumpHandler()* method defined inside the *CBasicCommandsModule* class. The first parameter of this method is a pointer to the C++ *type_info* structure, which contains type information about the object for which the handler is to be registered. This structure can be obtained by using the C++ *typeid* operator. The second parameter is a pointer to the instance of the handler, which should be registered. Detailed information about the *registerDumpHandler()* method can be found inside the API documentation.

Example:

```
#include "IDumpObjectHandler.h"
#include "CBasicCommandsModule.h"

// Simple list handler class
class CDumpSkeletonHandler : public IDumpObjectHandler
{
      void dumpObject(IObject* object)
      {
            // Implementation of handler goes here
      }
};

// Get the pointer to the instance of the basic commands module
CBasicCommandsModule* module = CBasicCommandsModule::getInstance();

// Create an instance of the CDumpSkeletonHandler class
CDumpSkeletonHandler DumpSkeletonHandler

// Register handler for the object type CSkeleton
module->registerDumpHandler(&typeid(CSkeleton),&DumpSkeletonHandler);
```

This example defines a handler class, creates an instance of the class and registers this instance as a dump handler.

Of course, a handler can be unregistered as well by calling the *unregisterDumpHandler()* method of the *CBasicCommandsModule* class and passing a pointer to the C++ *type_info* structure of the object, for which the handler is to be removed.

NOTE: The instances of registered handlers are not destroyed automatically by the framework. This must be done explicitly.