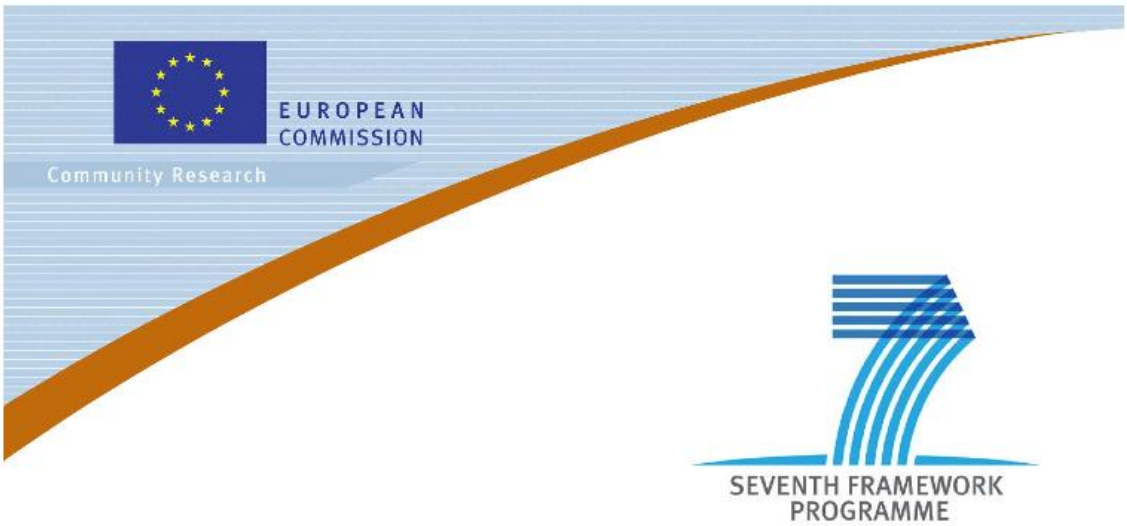


Private Public Partnership Project (PPP)

Large-scale Integrated Project (IP)

D.13.4.3: FI-WARE User and Programmers Guide

Project acronym: FI-WARE
Project full title: Future Internet Core Platform
Contract No.: 285248
Strategic Objective: FI.ICT-2011.1.7 Technology foundation: Future Internet Core Platform
Project Document Number: ICT-2011-FI-285248-WP13-D.13.4.3
Project Document Date: 2014-04-30
Deliverable Type and Security: Public
Author: FI-WARE Consortium
Contributors: FI-WARE Consortium

1.1 Executive Summary

This document describes the usage of each Generic Enabler provided by the "Advanced Middleware and Web-based User Interfaces" chapter. The document also details the required software elements and the procedure to setup an environment for the development of applications that can exploit the capabilities of the Advanced Middleware and Web-based User Interfaces Generic Enablers..

1.2 About This Document

This document comes along with the Software implementation of components, each release of the document being referred to the corresponding Software release (as per D.x.2), to provide documentation of the features offered by the components and interfaces to users/adopters. Moreover, it explains the way they can be exploited in their developments.

1.3 Intended Audience

The document targets users as well as programmers of FI-WARE Generic Enablers.

1.4 Chapter Context

FI-WARE will enable smarter, more customized/personalized and context-aware applications and services by the means of a set of assets able to gather, exchange, process and analyze massive data in a fast and efficient way. Nowadays, several well-known free Internet services are based on business models that exploit massive data provided by end users. This data is exploited in advertising or offered to 3rd parties so that they can build innovative applications. Twitter, Facebook, Amazon, Google and many others are examples of this.

The Advanced Middleware and Web User Interface (UI) Architecture chapter (aka. MiWi) of FI-WARE offers Generic Enablers from two different but related areas:

Advanced Middleware

The high-performance middleware that is backward compatible with traditional Web services (e.g. REST) but offers advanced features and performance and dynamically adapts to the communication partners and its environment. A novel API separates WHAT data needs to be communicated from WHERE the data come from within the native data structures of the application, and HOW the data should be transmitted to the target. Additionally, the middleware offers "Security by Design" through a declarative API, where the application defines the security requirements and policies that apply to its data, which are then automatically enforced by the middleware. The middleware uses of the security functionality offered by the Security chapter of FI-WARE.

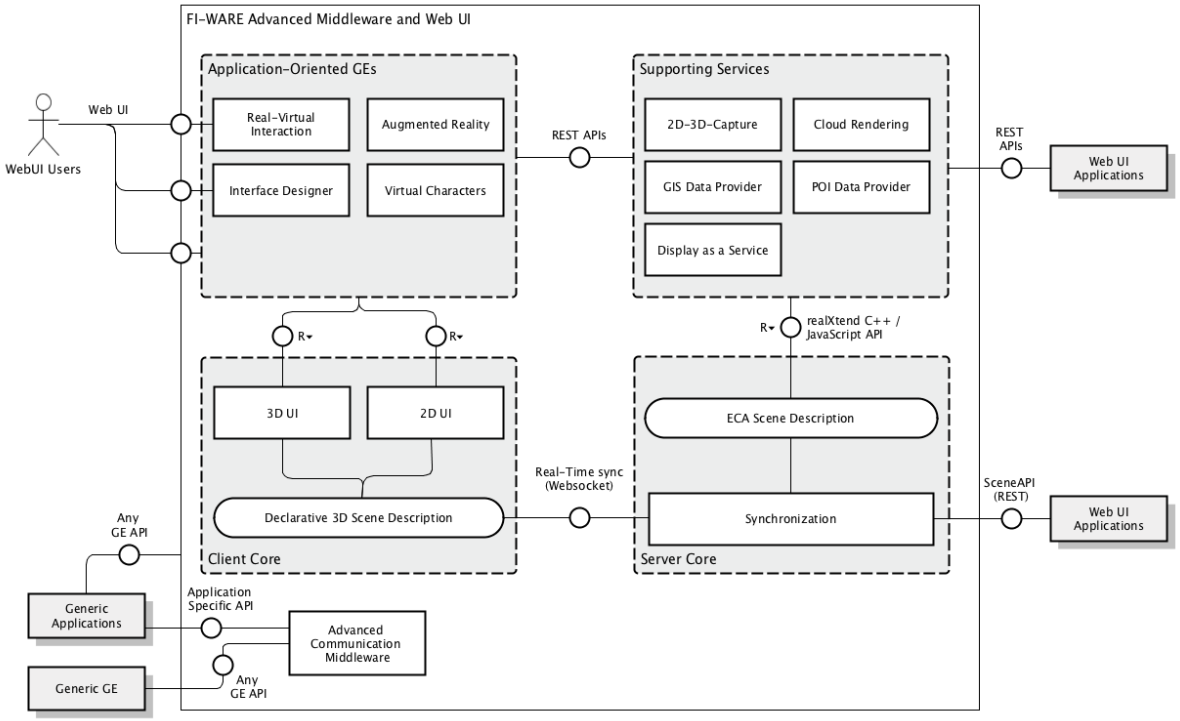
Advanced Web-based User Interface (Web UI)

In order to become widely visible and adopted by end users, the FI-WARE Future Internet platform must not only offer server functionality but must also offer much improved user experiences. The objective is to significantly improve the user experience for the Future Internet by adding new user input and interaction capabilities, such as interactive 3D graphics, immersive interaction with the real and virtual world (Augmented Reality), virtualizing and thus separating the display from the (mobile) computing device for ubiquitous operations, and many more. The technology is based on the Web technology stack, as the Web is quickly becoming THE user interface technology supported on essentially any (mobile) device while already offering advanced rich media capabilities (e.g. well-formatted text, images, video). First devices are becoming available that use Web technology even as the ONLY user interface technology. The Web design and programming environment is well-known to millions of developers that allow quick uptake of new technology, while offering a proven model for continuous and open innovation and improvement.

These two areas have been combined because highly interactive (2D/3D) user interfaces making use of service oriented architectures have strong latency, bandwidth, and performance requirements regarding the middleware implementations. An example is the synchronization service for real-time shared virtual worlds or the machine control on a factory floor that must use the underlying network and computing hardware as efficiently as possible. Generic Enables are provided to make optimal use of the underlying hardware via Software Defined Networking in the

Middleware (SDN, using the GEs of the Interface to Networks and Devices (I2ND) Architecture chapter) and the Hardware Support in the 3D-UI GE (see FIWARE.ArchitectureDescription.MiWi.3D-UI) that provides access to GPU and other parallel compute functionality that may be available. The following diagram shows the main components (Generic Enablers) that comprise the first release of FI-WARE Data/Context chapter architecture.

The following diagram shows the main components (Generic Enablers) that comprise FI-WARE Advanced Middleware and Web-based User Interfaces chapter architecture.



More information about the Advanced Middleware and Web-based UI Chapter and FI-WARE in general can be found within the following pages:

http://wiki.fi-ware.org

Advanced Middleware and Web UI Architecture

Materializing Advanced Middleware and Web User Interfaces in FI-WARE

1.5 Structure of this Document

The document is generated out of a set of documents provided in the public FI-WARE wiki. For the current version of the documents, please visit the public wiki at http://wiki.fi-ware.org/

The following resources were used to generate this document:

D.13.4.3_User_and_Programmers_Guide_front_page

Middleware - User and Programmers Guide

Middleware - KIARA - User and Programmers Guide

Middleware - RPC over DDS - User and Programmers Guide

Middleware - RPC over REST - User and Programmers Guide

Middleware - Fast Buffers - User and Programmers Guide

2D-UI - User and Programmers Guide

3D-UI - XML3D - User and Programmers Guide

3D-UI - WebTundra - User and Programmers Guide

Synchronization - User and Programmers Guide

Cloud Rendering - User and Programmers Guide

Display As A Service - User and Programmers Guide

GIS Data Provider - User and Programmers Guide

POI Data Provider - User and Programmers Guide

2D-3D Capture - User and Programmers Guide

Augmented Reality - User and Programmers Guide

Real Virtual Interaction - User and Programmers Guide

Virtual Characters - User and Programmers Guide

Interface Designer - User and Programmers Guide

1.6 Typographical Conventions

Starting with October 2012 the FI-WARE project improved the quality and streamlined the submission process for deliverables, generated out of our wikis. The project is currently working on the migration of as many deliverables as possible towards the new system.

This document is rendered with semi-automatic scripts out of a MediaWiki system operated by the FI-WARE consortium.

1.6.1 Links within this document

The links within this document point towards the wiki where the content was rendered from. You can browse these links in order to find the "current" status of the particular content.

Due to technical reasons part of the links contained in the deliverables generated from wiki pages cannot be rendered to fully working links. This happens for instance when a wiki page references a section within the same wiki page (but there are other cases). In such scenarios we preserve a link for readability purposes but this points to an explanatory page, not the original target page.

In such cases where you find links that do not actually point to the original location, we encourage you to visit the source pages to get all the source information in its original form. Most of the links are however correct and this impacts a small fraction of those in our deliverables.

1.6.2 Figures

Figures are mainly inserted within the wiki as the following one:

```
[[Image:....|size|alignment|Caption]]
```

Only if the wiki-page uses this format, the related caption is applied on the printed document. As currently this format is not used consistently within the wiki, please understand that the rendered pages have different caption layouts and different caption formats in general. Due to technical reasons the caption can't be numbered automatically.

1.6.3 Sample software code

Sample API-calls may be inserted like the following one.

```
http://[SERVER_URL]?filter=name:Simth*&index=20&limit=10
```

1.7 Acknowledgements

The current document has been elaborated using a number of collaborative tools, with the participation of Working Package Leaders and Architects as well as those partners in their teams they have decided to involve. The following partners contributed to this deliverable: ADMINO, CYBER, UOULU,LUDOCRAFT,PLAYSIGN, ZHAW, EPROS, USAAR-CISPA, DFKI.

1.8 Keyword list

FI-WARE, PPP, Architecture Board, Steering Board, Roadmap, Reference Architecture, Generic Enabler, Open Specifications, I2ND, Cloud, IoT, Data/Context Management, Applications/Services Ecosystem, Delivery Framework , Security, Developers Community and Tools , ICT, Middleware, 3D User Interfaces.

1.9 Changes History

Release	Major changes description	Date	Editor
v1	Initial Version	2014-06-23	ZHAW

1.10 Table of Contents

2 Middleware - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

2.1 Introduction

In contrast to other GEs, the FI-WARE Middleware GE is not a standalone service running in the network, but a set of compile-/runtime tools and a communication library to be delivered with the application. Not all applications have the same requirements regarding dynamic code integration and protocol/network negotiation. For this reason we deliver, beside the integrated full dynamic runtime KIARA middleware framework also other components, which serve basic middleware functionalities.

The **KIARA middleware suite** Release 3.3 consists of the following components:

KIARA framework

The KIARA framework is a middleware that realizes a novel approach for connecting application code with a framework. It supports currently remote procedure calls (RPC) but support for publish/subscribe communication is planned as well.

Instead of forcing an application to use data types predefined by the middleware (as e.g. in Thrift, CORBA, etc.) applications describes its own native data types to the KIARA middleware. The middleware then generates the necessary code to access those data structures at run-time using an embedded compiler. Because due to the negotiation of the optimal protocol for the targeted server, KIARA can generate the optimal code to serialize the native data structures for the chosen protocol.

This approach allows to combine KIARA with arbitrary applications without rewriting major parts of the application itself or adding redundant copy operations between the native and middleware generated data structures, as required by other middleware.

RPC over DDS

RPC over DDS is based on the Data Distribution Service (DDS) specifications, an OMG Standard defining the API and Protocol for high performance publish-subscribe middleware. eProsima RPC over DDS is an Remote Procedure Call framework using DDS as the transport and is based on the ongoing OMG RPC for DDS standard. In this release of the middleware for FI-WARE (R3.2), we are providing the basic assets, DDS and eProsima RPC for DDS, and other modules that are building blocks of the KIARA Middleware suite.

RPC over REST

One of the main FI-WARE middleware requirements is to offer backwards compatibility with Web Services, specifically RESTful Web Services.

eProsima RPC over REST enable the creation and the invocation of RESTful Web Services through the common API also used in eProsima RPC over DDS, and the future eProsima RPC over TCP, both part of the FI-WARE middleware suite.

FastBuffers & Dynamic Fast Buffers

eProsima Fast Buffers is an open source serialization engine optimized for performance, beating alternatives such as Apache Thrift and Google Protocol Buffers in both Simple and Complex Structures.

It generates serialization code for your structured data from its definition in an Interface Description Language (IDL).

2.2 User and Programmers Guides

For each of the components we provide separate User and Programmers Guides

- Middleware - KIARA - User and Programmers Guide
- Middleware - RPC over DDS - User and Programmers Guide
- Middleware - RPC over REST - User and Programmers Guide
- Middleware - Fast Buffers - User and Programmers Guide

3 Middleware - KIARA - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

3.1 Introduction

KIARA framework is a middleware that realizes a novel approach for connecting application code with a framework. It supports currently remote procedure calls (RPC) but support for publish/subscribe communication is planned as well.

Instead of forcing an application to use data types predefined by the middleware (as e.g. in Thrift, CORBA, etc.) applications describes its own native data types to the KIARA middleware. The middleware then generates the necessary code to access those data structures at run-time using an embedded compiler. Because due to the negotiation of the optimal protocol for the targeted server, KIARA can generate the optimal code to serialize the native data structures for the chosen protocol.

This approach allows to combine KIARA with arbitrary applications without rewriting major parts of the application itself or adding redundant copy operations between the native and middleware generated data structures, as required by other middleware.

This manual describes C and C++ language support of the KIARA middleware.

3.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Advanced Middleware GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

3.2 User Guide

KIARA is a developer tool. This component is for programmers, who will invoke the APIs programmatically and there is no user interface as such.

3.3 Programmers Guide

3.3.1 IDL Types

The interface definition language (IDL) is used to describe interfaces of the services provided at the remote endpoint, where remote endpoint not necessarily means a different host. The service could even exist as a plug-in within the same application. IDL uses data types and structures, but is platform-, architecture-, and application-agnostic.

The basic KIARA IDL types are based on the Thrift types: http://thrift.apache.org/docs/types/. These types are distinct from native application types and used solely to describe abstract service interfaces.

3.3.1.1 *Primitive types*

KIARA provides following primitive types:

- boolean: A boolean value (true or false)
- i8: An 8-bit signed integer
- u8: An 8-bit unsigned integer
- i16: A 16-bit signed integer
- u16: A 16-bit unsigned integer

- i32: A 32-bit signed integer
- u32: A 32-bit unsigned integer
- i64: A 64-bit signed integer
- u64: A 64-bit unsigned integer
- float: A 32-bit floating point number
- double: A 64-bit floating point number
- string: A text string encoded using UTF-8 encoding

In the following table KIARA types are compared with types used in other systems:

KIARA	CORBA	.NET	Thrift	SOAP	WebIDL	Range
boolean	boolean	Boolean	bool	boolean	boolean	{true, false}
i8		SByte	byte	byte	byte	[-128, 127]
u8	octet	Byte		unsignedByte	octet	[0, 255]
i16	short	Int16	i16	short	short	[-32768, 32767]
u16	unsigned short	UInt16		unsignedShort	unsigned short	[0, 65535]
i32	long	Int32	i32	int	long	[-2147483648, 2147483647]
u32	unsigned long	UInt32		unsignedInt	unsigned long	[0, 4294967295]
i64	long long	Int64	i64	long	long long	[-9223372036854775808, 9223372036854775807]
u64	unsigned long long	UInt64		unsignedLong	unsigned long long	[0, 18446744073709551615]
float	float	Single	float	float	float	http://en.wikipedia.org/wiki/Single_precision
double	double	Double	double	double	double	http://en.wikipedia.org/wiki/Double-precision_floating-point_format
string	string	String	string	string	DOMString	

3.3.1.2 *Structs*

KIARA structs define a derived type that represent a collection of members similar to C/C++'s struct. A struct has a set of strongly typed fields, each with a unique name identifier. Fields may have various annotations (numeric field IDs, optional default values, etc.) that are described in the KIARA IDL.

3.3.1.3 *Containers*

KIARA supports following container types:

- `array< Type, Size >` - fixed length arrays (e.g. `array< i32, 4 >`)
- `array< Type >` - dynamic arrays

Multidimensional arrays can be constructed by combining multiple `array`'s: 4x4 float array - `array< array< float, 4>, 4>`

3.3.1.4 *Exceptions*

Exceptions are functionally equivalent to structs and are thrown when service functions fails.

3.3.1.5 *Services*

Services are similar to structs, but contain only functions.

3.3.1.6 *Functions*

Functions can only be members of services. Each function has return type, list of named arguments, and list of named exceptions. Function, its arguments and exceptions can be annotated. Functions can have `void` type as a return type.

3.3.1.7 *Annotations*

Annotations are functionally equivalent to structs but are only used to represent additional information to the IDL.

3.3.2 Describing Application Data Types

In order to connect remote services, abstract IDL types described above need to be mapped to the application-specific data types. KIARA API provides a novel method to perform this task in a non-intrusive way. Instead of forcing to use predefined data types of the middleware (e.g. Thrift, CORBA, etc.) applications describes their own data types. This allows to combine KIARA with arbitrary applications without rewriting major parts of the application itself.

Application data types of C/C++ application are described with macros which create static in memory representation of the data types. Macros are used as they provide a type-safe way of connecting the type descriptions to the data structures they describe such that changes will automatically detected. These macros mostly declare static data structures than are later references when interface functions and services are declared. Macros can be written by hand, which can be somewhat tedious especially for C language, or generated automatically with the KIARA preprocessor tool.

3.3.2.1 *Preprocessor*

The `kiara-preprocessor` tool provides a convenient way to describe application data types to be used with KIARA framework. Input of the the tool is the application source code and a number of simple annotations that describe which types should be used with KIARA. These macros can be either embedded directly into application source code or can be provided in a separate header file.

3.3.2.1.1 *Describing primitive data types*

In our example we will use following IDL:

```
namespace * calc
service calc {
    i32 add(i32 a, i32 b)
    float addf(float a, float b)
    i32 stringToInt32(string s)
    string int32ToString(i32 i)
}
```

For the client we need to generate functions that will perform a remote call and connect them with the remote methods of the service:

```
kiara_declare_func(Calc_Add, int & result_value result, int a, int b)
kiara_declare_func(Calc_Add_Float, float & result_value result, float
a, float b)
kiara_declare_func(Calc_String_To_Int32, int & result_value result,
const char *s)
kiara_declare_func(Calc_Int32_To_String, char ** result_value result,
int i)
```

kiara_declare_func(*type_name*, ...) macro declares remote function that can be called by the client. All arguments after the function name are argument types and names to the function. **result_value** macro can be used between type and name argument for describing arguments that are received as a result of the call (output arguments). Note that **result_value** can only be applied to a pointer or reference type, which can receive a result value.

For the server we need to generate service functions that will be called upon remote call:

```
kiara_declare_service(Calc_Add_Impl, int & result_value result, int a,
int b)
kiara_declare_service(Calc_Add_Float_Impl, float & result_value result,
float a, float b)
kiara_declare_service(Calc_String_To_Int32_Impl, int & result_value
result, const char *s)
kiara_declare_service(Calc_Int32_To_String_Impl, char ** result_value
result, int i)
```

kiara_declare_service(*type_name*, ...) macro is similar to the **kiara_declare_func** and declares service function that can be called by the client. All arguments after the function name are argument types and names to the function.

Running `kiara-preprocessor` tool on the source code with these macros will generate KIARA API macros that will describe not just the client and server function but also generates macros for all involved data types.

3.3.2.1.2 *Describing complex data types*

The above described approach works also for C/C++ structs as long as they contain combination of structs and primitive data types like int, float, etc. More complex data types require additional annotation since the preprocessor cannot extract the necessary application semantics automatically (C/C++ are low-level languages that do not provide all the needed semantics

implicitly). Such types can be described by providing custom user functions which will access the internals of a type. We call such types *opaque*.

For example C++ `std::string` can be mapped to IDL `string` object, but in order to do this KIARA need to know how to get and set internal character sequence:

```
int std_string_SetCString(KIARA_UserType *ustr, const char *cstr)
{
    if (cstr)
        ((std::string*)ustr)->assign(cstr);
    else
        ((std::string*)ustr)->clear();
    return 0;
}

int std_string_GetCString(KIARA_UserType *ustr, const char **cstr)
{
    *cstr = ((std::string*)ustr)->c_str();
    return 0;
}

kiara_declare_opaque_object(std::string,
                            kiara_user_api(SetCString,
std_string_SetCString),
                            kiara_user_api(GetCString,
std_string_GetCString))
```

kiara_declare_opaque_object(type_name, user_api_entries) macro associates multiple custom user functions with API known to KIARA. In our case we register `std_string_GetCString` as a getter and `std_string_SetCString` as a setter functions which return and set std::string contents as a C's null-terminated string respectively. `SetCString` and `GetCString` are reserved KIARA keywords that describe an API functions that the user implements.

With this additional declaration we can use `std::string` when calling and implementing service methods `stringToInt32` and `int32ToString`:

```
// for client
kiara_declare_func(Calc_StdString_To_Int32, int & result_value result,
const std::string & s)
kiara_declare_func(Calc_Int32_To_StdString, std::string & result_value
result, int i)

// for server
kiara_declare_service(Calc_StdString_To_Int32_Impl, int & result_value
result, const std::string & s)
```

```
kiara_declare_service(Calc_Int32_To_StdString_Impl, std::string &
result_value result, int i)
```

Besides manipulating contents of opaque types KIARA need to properly allocate and deallocate them:

```
KIARA_UserType * std_string_Allocate(void)
{
    return (KIARA_UserType *)new std::string;
}

void std_string_Deallocate(KIARA_UserType *value)
{
    delete (std::string*)value;
}

kiara_declare_opaque_object(std::string,
                            kiara_user_api(SetCString,
std_string_SetCString),
                            kiara_user_api(GetCString,
std_string_GetCString),
                            kiara_user_api(AllocateType,
std_string_Allocate),
                            kiara_user_api(DeallocateType,
std_string_Deallocate))
```

`AllocateType` and `DeallocateType` represent APIs for allocating and deallocating type respectively.

3.3.2.1.3 *Array members in structure declaration*

Besides opaque types we also may need to describe more complex structure members like C-arrays. They are often defined in C and sometimes in C++ structs as a combination of an integer member representing array size and a pointer member pointing to the first element of the array. For example:

```
typedef struct IntArray
{
    int size;
    int *array;
} IntArray;
```

Such member combination can be described with the preprocessor macro **kiara_struct_array_member**:

```
/* KIARA declaration of IntArray */

kiara_declare_struct(IntArray,
```

```
    kiara_struct_array_member(array, size))
```

The **kiara_declare_struct(type_name, ...)** macro allows to override a type declaration which usually would otherwise be automatically generated.

3.3.2.2 *Static type construction*

The preprocessor described above generates source code containing KIARA API macros that need to be included in the application part that uses the KIARA API for calling services. These macros can be also written manually or maybe even generated by another tool. In this section we will describe these macros in more detail.

Internally these macros construct an internal chain of static data structures containing all information required for accessing user-defined data at the run-time. Macros for C are defined in the `KIARA/kiara_macros.h` header, and for C++ in `KIARA/kiara_cxx_macros.hpp`. For C++ user can choose to use either C or C++ macros. However they cannot be mixed.

3.3.2.2.1 *Primitive types*

For declaring C-type for KIARA it must have a name, for example by defining it with a typedef. KIARA predefines names for all primitive C types:

KIARA macro	C type
KIARA_CHAR	char
KIARA_WCHAR_T	wchar_t
KIARA_SCHAR	signed char
KIARA_UCHAR	unsigned char
KIARA_SHORT	short
KIARA_USHORT	unsigned short
KIARA_INT	int
KIARA_UINT	unsigned int
KIARA_LONG	long
KIARA_ULONG	unsigned long
KIARA_LONGLONG	long long
KIARA_ULONGLONG	unsigned long long
KIARA_SIZE_T	size_t

KIARA_SSIZE_T	ssize_t
KIARA_VOID	void
KIARA_FLOAT	float
KIARA_DOUBLE	double
KIARA_LONGDOUBLE	long double
KIARA_CHAR_PTR	char *
KIARA_VOID_PTR	void *
KIARA_INT8_T	int8_t
KIARA_UINT8_T	uint8_t
KIARA_INT16_T	int16_t
KIARA_UINT16_T	uint16_t
KIARA_INT32_T	int32_t
KIARA_UINT32_T	uint32_t
KIARA_INT64_T	int64_t
KIARA_UINT64_T	uint64_t

Note: this list contains all primitive types that can be described with KIARA, not all of them can be directly mapped to the IDL types. See Calling remote functions section for supported mappings.

3.3.2.2.2 *Pointer declaration*

KIARA supports pointer declaration with **KIARA_DECL_PTR** macro.

```
/* Note: KIARA_INT and KIARA_FLOAT are predefined, float* is not */
typedef float * FloatPtr;

/* int pointer */
KIARA_DECL_PTR(IntPtr, KIARA_INT)

/* float pointer */
KIARA_DECL_PTR(FloatPtr, KIARA_FLOAT)
```

```
/* FloatPtr* */
KIARA_DECL_PTR(FloatPtrPtr, FloatPtr)
```

KIARA_DECL_PTR(*ptr_type_name*, *element_type_name*) macro declare a pointer type for the existing named C-type specified as the second argument. Const pointers are declared with **KIARA_DECL_CONST_PTR** macro:

```
/* const int pointer */
KIARA_DECL_CONST_PTR(ConstIntPtr, KIARA_INT)
```

In C++ pointer declaration is not needed.

3.3.2.2.3 *Structure declaration*

Structures are declared with the **KIARA_DECL_STRUCT** macro:

```
/* User-defined C struct */

typedef struct {
    float x;
    float y;
} Vec2f;

/* KIARA declaration of Vec2f */

KIARA_DECL_STRUCT(Vec2f,
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, x)
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, y)
)

/* User-defined C struct */

typedef struct {
    Vec2f a;
    Vec2f b;
} Linef;

/* KIARA declaration of Linef */

KIARA_DECL_STRUCT(Linef,
  KIARA_STRUCT_MEMBER(Vec2f, a)
  KIARA_STRUCT_MEMBER(Vec2f, b)
)
```

KIARA_DECL_STRUCT(*type_name*, *members*) macro expects two arguments: name of a C struct type and a sequence of **KIARA_STRUCT_MEMBER** macros. All KIARA macros require that passed type names are typedef names, not tag names. **KIARA_STRUCT_MEMBER(*member_type_name*, *member_name*)** macro expects also two arguments describing struct member: name of a member type and name of a member. Note that sequence is **not** separated by comma. Also it is allowed to omit members in the described structure.
The type name passed to **KIARA_STRUCT_MEMBER** macro must either be defined previously by one of the KIARA declaration macros or be one of predefined for primitive C types.
When using C++ alternative macros **KIARA_CXX_DECL_STRUCT** and **KIARA_CXX_STRUCT_MEMBER** can be used instead:

```
/* KIARA declaration of Vec2f */

KIARA_CXX_DECL_STRUCT(Vec2f,
  KIARA_CXX_STRUCT_MEMBER(x)
  KIARA_CXX_STRUCT_MEMBER(y)
)

/* KIARA declaration of Linef */

KIARA_CXX_DECL_STRUCT(Linef,
  KIARA_CXX_STRUCT_MEMBER(a)
  KIARA_CXX_STRUCT_MEMBER(b)
)
```

Because internally **KIARA_CXX_DECL_STRUCT** uses C++ templates, types of the struct members can be deduced automatically.

3.3.2.2.4 *Array members in structure declaration*

Arrays are often defined in C and sometimes in C++ structs as a combination of an integer member representing array size and a pointer member pointing to the first element of the array. For example:

```
typedef struct IntArray
{
    int size;
    int *array;
} IntArray;
```

Such member combination can be described with the KIARA macro **KIARA_STRUCT_ARRAY_MEMBER**:

```
/* KIARA declaration of IntArray */

KIARA_DECL_PTR(IntPtr, KIARA_INT)

KIARA_DECL_STRUCT(IntArray,
```

```
  KIARA_STRUCT_ARRAY_MEMBER(IntPtr, array, KIARA_INT, size)
)
```

KIARA_STRUCT_ARRAY_MEMBER(*ptr_member_type_name*, *ptr_member_name*, *size_member_type_name*, *size_member_name*) macro expects also four arguments describing array member: name of a pointer type, name of a pointer member, name of a size type, and name of a size member.
The type name passed to **KIARA_STRUCT_ARRAY_MEMBER** macro must either be defined previously by one of the KIARA declaration macros or be one of predefined for primitive C types.
Arrays are allocated by default with **malloc** and deallocated with **free** C function calls.
When using C++ alternative macro **KIARA_CXX_STRUCT_ARRAY_MEMBER** can be used instead:

```
/* KIARA declaration of IntArray */


KIARA_CXX_DECL_STRUCT(IntArray,
  KIARA_CXX_STRUCT_ARRAY_MEMBER(array, size)
)
```

Because internally **KIARA_CXX_STRUCT_ARRAY_MEMBER** uses C++ templates, types of the struct members can be deduced automatically.

3.3.2.2.5 *Function declaration*

Signatures of functions that are generated by KIARA are declared with the **KIARA_DECL_FUNC** macro:

```
/* KIARA declaration of a float* type */
KIARA_DECL_PTR(FloatPtr, KIARA_FLOAT)


/* KIARA declaration of a function with signature:


   typedef int (*Func)(float *result, int a, float b, double c);
 */


KIARA_DECL_FUNC(Func,
  KIARA_FUNC_RESULT(FloatPtr, result)
  KIARA_FUNC_ARG(KIARA_INT, a)
  KIARA_FUNC_ARG(KIARA_FLOAT, b)
  KIARA_FUNC_ARG(KIARA_DOUBLE, c)
)
```

KIARA_DECL_FUNC(*type_name*, *func_args*) macro is similar to **KIARA_DECL_STRUCT** and expects two arguments: name of a C function type and sequence of `KIARA_FUNC_ARG` and `KIARA_FUNC_RESULT` macros. **KIARA_FUNC_ARG(*type_name*, *member_name*)** macro expects two arguments describing function argument: name of an argument type and a name of an argument. **KIARA_FUNC_RESULT(*type_name*, *member_name*)** does the same as **KIARA_FUNC_ARG**, but additionally marks the function parameter as a result of a function call. The return type of the declared function signature is always int and is used to report errors. Note that sequence is **not** separated by comma. The type name passed to `KIARA_FUNC_ARG` and

`KIARA_FUNC_RESULT` macros must either be defined previously by one of the KIARA declaration macros or be one of the predefined for primitive C types (see above).
Similar to structs functions can be declared in C++ with **KIARA_CXX_DECL_FUNC**, **KIARA_CXX_FUNC_ARG**, and **KIARA_CXX_FUNC_RESULT**:

```
/* KIARA declaration of Func */

KIARA_CXX_DECL_FUNC(Func,
  KIARA_CXX_FUNC_RESULT(float *, result)
  KIARA_CXX_FUNC_ARG(int, a)
  KIARA_FUNC_ARG(float, b)
  KIARA_FUNC_ARG(double, c)
)
```

In C++ derived types of function parameters like pointers and references do not need to be explicitly declared.

3.3.2.2.6 *Array declaration*
Arrays are declared with **KIARA_DECL_ARRAY**, **KIARA_DECL_FIXED_ARRAY**, and **KIARA_DECL_FIXED_ARRAY_2D** macros.

```
typedef int IntArray[];
typedef float FloatArray4[4];
typedef double DoubleMatrix4[4][4];

/* Unbounded array declaration */

KIARA_DECL_ARRAY(IntArray, KIARA_INT)

/* Fixed size 1D array declaration */

KIARA_DECL_FIXED_ARRAY(FloatArray4, KIARA_FLOAT, 4)

/* Fixed size 2D array declaration */

KIARA_DECL_FIXED_ARRAY_2D(mat44, KIARA_FLOAT, 4, 4)
```

KIARA_DECL_ARRAY(*array_type_name*, *element_type_name*) macro declares one-dimensional unbounded arrays. **KIARA_DECL_FIXED_ARRAY(*array_type_name*, *element_type_name*, *size*)** macro declares one-dimensional arrays with fixed length. **KIARA_DECL_FIXED_ARRAY_2D(*array_type_name*, *element_type_name*, *num_rows*, *num_cols*)** macro declares two-dimensional arrays with fixed length.

All macros expect name of the declared type as the first argument, previously declared name of the array element as the second, and finally dimension values.

In C++ array types do not need to be explicitly declared.

3.3.2.2.7 *Forward declaration*

KIARA supports forward declaration of types which is required for declaring cyclic structures. KIARA type is forward declared with **KIARA_FORWARD_DECL** macro.

```
typedef struct IntList {
    struct IntList *next;
    int data;
} IntList;

/* Forward declaration of a named type */
KIARA_FORWARD_DECL(IntList)

/* Pointer declaration */
KIARA_DECL_PTR(IntListPtr, IntList)

KIARA_DECL_STRUCT(IntList,
  KIARA_STRUCT_MEMBER(IntListPtr, next)
  KIARA_STRUCT_MEMBER(KIARA_INT, data)
)
```

KIARA_FORWARD_DECL(*type_name*) macro performs a forward declaration of a specified type.

In C++ forward declaration is not needed.

3.3.2.2.8 *Opaque types*

Often data structures are part of the implementation and are not defined in a public API. Such data structures are called abstract data types (ADT). In C and C++ such data structures are represented by a declaration of a struct type without public definition. Such struct types are called opaque, and the only way to access their data is to use a public API provided along with the opaque type.

Data access

In KIARA opaque type is defined with **KIARA_DECL_OPAQUE_TYPE** macro. In order to access its contents KIARA needs to know how to get and set its internal data. The access is performed by getter and setter API functions which signature is known to KIARA. These functions are registered with the opaque type by using **KIARA_USER_API** macro. KIARA recursively processes an opaque type by calling API function until it reaches a primitive type supported by default.

In the following example we describe provided by KIARA `kr_dstring_t` abstract data type that implements dynamic strings in C:

```
int dstring_SetCString(KIARA_UserType *ustr, const char *cstr)
{
    int result = kr_dstring_assign_str((kr_dstring_t*)ustr, cstr);
    return result ? 0 : 1;
}
```

```
int dstring_GetCString(KIARA_UserType *ustr, const char **cstr)
{
    *cstr = kr_dstring_str((kr_dstring_t*)ustr);
    return 0;
}

KIARA_DECL_OPAQUE_TYPE(kr_dstring_t,
       KIARA_USER_API(SetCString, dstring_SetCString)
       KIARA_USER_API(GetCString, dstring_GetCString))
```

In our example contents of a `kr_dstring_t` type can be returned as a C-string and set to a C-string. Thus we implement and register two API access functions named `SetCString` and `GetCString` for setting and getting C-strings from a `kr_dstring_t` type respectively.

With C++ API we can describe `std::string` type in a similar way:

```
static int std_string_SetCString(KIARA_UserType *ustr, const char
*cstr)
{
    if (cstr)
        ((std::string*)ustr)->assign(cstr);
    else
        ((std::string*)ustr)->clear();
    return 0;
}

static int std_string_GetCString(KIARA_UserType *ustr, const char
**cstr)
{
    *cstr = ((std::string*)ustr)->c_str();
    return 0;
}

KIARA_CXX_DECL_OPAQUE_TYPE(std::string,
       KIARA_CXX_USER_API(SetCString, std_string_SetCString)
       KIARA_CXX_USER_API(GetCString, std_string_GetCString))
```

3.3.2.2.9 *Memory management*

Besides accessing contents of an opaque type KIARA needs to know how to allocate and deallocate a type. By default KIARA allocates a type by calling C's `malloc` function. However, when type require additional initialization like in the case with `std::string` and `kr_dstring_t`, we need to register custom allocation and deallocation functions with the opaque type.

```
static KIARA_UserType * std_string_Allocate(void)
{
```

```
    return (KIARA_UserType *)new std::string;
}

void std_string_Deallocate(KIARA_UserType *value)
{
    delete (std::string*)value;
}

KIARA_CXX_DECL_OPAQUE_TYPE(std::string,
       KIARA_CXX_USER_API(SetCString, std_string_SetCString)
       KIARA_CXX_USER_API(GetCString, std_string_GetCString)
       KIARA_CXX_USER_API(AllocateType, std_string_Allocate)
       KIARA_CXX_USER_API(DeallocateType, std_string_Deallocate))
```

3.3.2.2.10 *Structs with API*

Not only opaque types, but usual structures may require custom access or allocation behavior. For example when members of a structure require additional initialization after allocation. For this case API functions can be registered for structs:

```
typedef struct Data
{
    int ival;
    kr_dstring_t sval;
} Data;

void initData(Data *data)
{
    data->ival = 0;
    kr_dstring_init(&data->sval);
}

void destroyData(Data *data)
{
    kr_dstring_destroy(&data->sval);
}

KIARA_UserType * Data_Allocate(void)
{
    Data *data = malloc(sizeof(Data));
    initData(data);
    return (KIARA_UserType *)data;
```

```
}


void Data_Deallocate(KIARA_UserType *value)
{
    if (value)
    {
        destroyData((Data*)value);
        free(value);
    }
}



KIARA_DECL_STRUCT_WITH_API(Data,
  KIARA_STRUCT_MEMBER(KIARA_INT, ival)
  KIARA_STRUCT_MEMBER(kr_dstring_t, sval),
  KIARA_USER_API(AllocateType, Data_Allocate)
  KIARA_USER_API(DeallocateType, Data_Deallocate)
)
```

The `Data` structure need to be initialized after allocation and uninitialized after deallocation. Thus it requires custom allocation and deallocation functions.

3.3.2.2.11 *Service declaration*

Similarly to functions generated by KIARA for calling remote services, a remote service function can be declared as well.

```
KIARA_DECL_SERVICE(Add,
    KIARA_SERVICE_RESULT(IntPtr, result)
    KIARA_SERVICE_ARG(KIARA_INT, a)
    KIARA_SERVICE_ARG(KIARA_INT, b))
{
    *result = a + b;
    return KIARA_SUCCESS;
}
```

KIARA_DECL_SERVICE(*function_name*, *func_args*) macro declares a function which can be called remotely, similarly to **KIARA_DECL_FUNC** macro.

3.3.3 Applications

3.3.3.1 ***Initialization and finalization***

Before KIARA library can be used it needs to be initialized with the call to `kiaraInit` function. After usage all allocated resources should be freed by a call to the `kiaraFinalize` function.

```
int main(int argc, char **argv)
```

```
{
    kiaraInit(&argc, argv);


    kiaraFinalize();
}
```

`kiaraInit` also processes command line options. Every KIARA option starts with `-kiara-`, currently the following options are supported:

Option	Description
-kiara-help	print list of all available command line options.
-kiara-config	specify configuration file
-kiara-module-path	module search path (by default value of environment variable KIARA_MODULE_PATH will be used)

All processed options are removed from the arguments list and argument count number is updated.

3.3.3.2 *Contexts*

KIARA require that each thread have a separate context that manages all internal KIARA data structures. Having separate context per thread allows to avoid explicit locking by the user. Context is created with `kiaraNewContext` function and destroyed with `kiaraFreeContext` function.

```
int main(int argc, char **argv)
{
    KIARA_Context *ctx;

    /* Initialize KIARA */
    kiaraInit(&argc, argv);

    /* Create context */
    ctx = kiaraNewContext();

    /* Destroy context */
    kiaraFreeContext(ctx);

    /* Finalize KIARA */
    kiaraFinalize();
}
```

Most of KIARA API functions require context as an argument.

3.3.3.3 *Establishing connections*

Before KIARA can call remote functions a connection to the remote KIARA node needs to be established with the call to `kiaraOpenConnection` function.

```
  KIARA_Connection *conn;
  KIARA_Context *ctx;

  /* Create context */
  ctx = kiaraNewContext();
  /* Open connection to the service */
  conn = kiaraOpenConnection(ctx, "http://myhost:80/service");
```

The URL used as the argument should refer to the service description which describes resources available at the endpoint of the connection. Currently service description has following structure:

```
{
    // Description of the server, optional
    info : "test server",

    // absolute or relative URL of the KIARA IDL
    idlURL : "/idl/calc.kiara",

    // KIARA IDL contents as string (either idlURL or idlContents
should be present)
    idlContents : "text",

    // List of servers providing services
    servers : [
        {
            // names of services that served with this configuration,
            // or "*" for all services
            services : "pattern",
            // specification of used marshalling protocol
            protocol : {
                name : "jsonrpc" // name of the protocol
            },
            // specification of used transport protocol
            transport : {
                // name of the transport protocol
                name : "http",
                // URL for URL-based transport
                url : "/rpc/calc"
```

```
            }
        },
        ...
    ]
}
```

In the process of establishing connection KIARA negotiates compatible protocols and transport methods and selects the best one supported by both KIARA nodes. Finally descriptions of services available on the remote side are fetched and added to the internal KIARA Type Description (KTD) managed by KIARA.

Connection is closed by calling to `kiaraCloseConnection`.

```
kiaraCloseConnection(conn);
```

3.3.3.4 *Calling remote functions*

In order to call a remote function KIARA generates a client stub function that accepts native application datatypes, serializes them to the format used by the current protocol, performs a call, and deserializes received response. For generating a client stub function **KIARA_GENERATE_CLIENT_FUNC(*connection*, *idl_method_name*, *func_type_name*, *mapping*)** macro is used. The arguments have the following meaning:

connection	- valid KIARA_Connection handle opened with **kiaraOpenConnection**.
idl_method_name	- name of the remote service method specified in the IDL.
func_type_name	- name of the function object type declared with the **KIARA_FUNC_OBJ(func_type_name)** macro.
mapping	- optional mapping of the IDL types to the application types. By default 1:1 mapping by names and types is used. Note: mapping is not implemented yet.

Consider following simple IDL:

```
namespace * calc

service calc {
    i32 add(i32 a, i32 b)
}
```

The IDL method `calc.add` can be mapped to the following function prototype:

```
KIARA_DECL_FUNC(Calc_Add,
  KIARA_FUNC_RESULT(IntPtr, result)
  KIARA_FUNC_ARG(KIARA_INT, a)
  KIARA_FUNC_ARG(KIARA_INT, b))
```

Basic types are mapped accordingly to the following table:

KIARA IDL Type	Default native type (C99 types)	Linux 32-bit C-Type

boolean	int32_t	int
i8	int8_t	char
u8	uint8_t	unsigned char
i16	int16_t	short
u16	uint16_t	unsigned short
i32	int32_t	int
u32	uint32_t	unsigned int
i64	int64_t	long long
u64	uint64_t	unsigned long long
float	float	float
double	double	double
string	char *	char *

Native types that are not in the table are mapped by converting them automatically to the required type. **Note: This feature is not implemented yet.**

The function instance performing call to the remote side is created in the following way:

```
KIARA_FUNC_OBJ(Calc_Add) add;


add = KIARA_GENERATE_CLIENT_FUNC(conn, "calc.add", Calc_Add, "");
```

The macro **KIARA_FUNC_OBJ(*type_name*)** defines function object type that can store stub instances of a given type generated by the **KIARA_GENERATE_CLIENT_FUNC** macro. The function object can be executed with the **KIARA_CALL** macro:

```
int result, errorCode;
errorCode = KIARA_CALL(add, &result, 21, 32);
```

Returned value is **KIARA_SUCCESS** on success and error code otherwise. Error code description can be retrieved by a call to the **kiaraGetConnectionError** function.

```
if (errorCode != KIARA_SUCCESS)
  fprintf(stderr, "Error: call failed: %s\n",
kiaraGetConnectionError(conn));
else
```

```
  printf("calc.add: result = %i\n", result);
```

In C++ a less complex syntax for defining and calling functions can be used.

```
KIARA_CXX_DECL_FUNC(Calc_Add,
  KIARA_CXX_FUNC_RESULT(int &, result)
  KIARA_CXX_FUNC_ARG(int, a)
  KIARA_CXX_FUNC_ARG(int, b))
```

Since C++ provides references we don't need to pass address of a result variable when calling `Calc_Add` function:

```
int result;
Calc_Add add = KIARA_GENERATE_CLIENT_FUNC(conn, "calc.add", Calc_Add,
"");
int errorCode = add(result, 21, 32);
```

3.3.3.5 *Defining remote services*

On the server side we need to define service functions that can be called. First, a service type is defined similarly to the function declaration:

```
KIARA_DECL_SERVICE(Calc_Add,
    KIARA_SERVICE_RESULT(IntPtr, result)
    KIARA_SERVICE_ARG(KIARA_INT, a)
    KIARA_SERVICE_ARG(KIARA_INT, b))
```

This declares only a prototype of the service function. Implementation of the service function must have the same number and type for all arguments. Additionally first argument must be of type **KIARA_ServiceFuncObj *** and provides information about connection. Result type must be **KIARA_Result**.

```
KIARA_Result calc_add_impl(KIARA_ServiceFuncObj *kiara_funcobj, int
*result, int a, int b)
{
    *result = a + b;
    return KIARA_SUCCESS;
}
```

When there is only a single implementation of the service function, both declarations can be combined:

```
KIARA_DECL_SERVICE_IMPL(Calc_Add,
    KIARA_SERVICE_RESULT(IntPtr, result)
    KIARA_SERVICE_ARG(KIARA_INT, a)
    KIARA_SERVICE_ARG(KIARA_INT, b))
{
    *result = a + b;
    return KIARA_SUCCESS;
}
```

When calling service function KIARA will automatically allocate memory, deserialize input, and serialize output parameters.

After defining service functions we need to create services, load IDL, and register service functions with the IDL methods:

```
KIARA_Context *ctx;
KIARA_Connection *conn;
KIARA_Service *service;
KIARA_Result result;

/* Create context */
ctx = kiaraNewContext();

/* Create service */
service = kiaraNewService(ctx);

/* Load and register IDL with the service */
result = kiaraLoadServiceIDLFromString(service,
    "KIARA",
    "namespace * calc "
    "service calc { "
    "    i32 add(i32 a, i32 b) "
    "} ");

/* Register calc.add IDL method with the Calc_Add service function */
result = KIARA_REGISTER_SERVICE_FUNC(service, "calc.add", Calc_Add, "",
calc_add_impl);
```

`kiaraLoadServiceIDLFromString` function parses IDL from string and registers it with the service, alternatively IDL can be loaded from file with `kiaraLoadServiceIDL` function. The first argument to `kiaraLoadServiceIDLFromString` is the service handle, then name of the IDL language, and finally string with the IDL.

KIARA_REGISTER_SERVICE_FUNC macro is similar to **KIARA_GENERATE_CLIENT_FUNC** macro, but just registers service function with the IDL method, specified by its full name. Internally KIARA will generate code that deserializes arguments, calls registered function, and finally serializes result and send it back to the caller. When combined declaration is used **KIARA_REGISTER_SERVICE_IMPL** needs to be used instead of **KIARA_REGISTER_SERVICE_FUNC**.

Finally we need to create server waiting for incoming connections, add all services that server should provide and run it.

```
server = kiaraNewServer(ctx, "0.0.0.0", 8080, "/service");

kiaraAddService(server, "/rpc/calc", "jsonrpc", service);
```

```
kiaraRunServer(server);


kiaraFreeServer(server);


kiaraFreeService(service);
```

`kiaraNewServer` function accepts host and port where connections will be accepted. Additionally an URL path with the location of the server configuration document is passed. Server configuration document is always delivered via HTTP protocol.

After server is created arbitrary number of services that should be served can be added with `kiaraAddService` function. `kiaraAddService` require server as the first argument, service path as second, protocol name as third and finally pointer to the service object created previously.

Service path describes transport method by which the service is provided. When path is relative, it is delivered with HTTP transport on the port which is specified with `kiaraNewServer` call. Absolute path contains a full specification of the transport method including host and port (e.g. http://0.0.0.0:8081/service/path or tcp://0.0.0.0:9090). Meaning of the path depends on the transport method. Currently supported are *HTTP* and *TCP* transports. When TCP transport is used only host and port from specified path are used.

Protocol is the name of the serialization protocol that is used to encode messages. Currently supported are *jsonrpc* (http://www.jsonrpc.org/specification), *tbp*, *ortecdr*, and *fastcdr*. Both *ortecdr* and *fastcdr* protocols use CDR encoding (http://en.wikipedia.org/wiki/Common_Data_Representation).

Finally server is started with the `kiaraRunServer` function.

3.3.4 Examples

3.3.4.1 *IDL*

We use following IDL in examples:

```
namespace * aostest


struct Vec3f {
 float x,
 float y,
 float z
}


struct Quatf {
 float r,
 Vec3f v
}


struct Location {
 Vec3f position,
 Quatf rotation
```

```
}

struct LocationList {
 array<Location> locations
}

service aostest {
  void setLocations(LocationList locations);
  LocationList getLocations();
}
```

3.3.4.2 *Common Source Code*

aostest_types.h - common data structures independent of KIARA framework

```
/*
 * This file contains application code and data structures independent
of KIARA framework
 */

#ifndef AOSTEST_TYPES_H_INCLUDED
#define AOSTEST_TYPES_H_INCLUDED

#include <KIARA/kiara.h>
#include <string.h>

#ifdef __cplusplus
extern "C" {
#endif

/* Location */

typedef struct Vec3f {
    float x;
    float y;
    float z;
} Vec3f;

typedef struct Quatf {
    float r; /* real part */
    Vec3f v; /* imaginary vector */
```

```
} Quatf;

typedef struct Location {
    Vec3f position;
    Quatf rotation;
} Location;

/* Data */

typedef struct LocationList
{
    int num_locations;
    Location *locations;
} LocationList;

static void initLocationList(LocationList *loclist, size_t size)
{
    loclist->num_locations = size;
    if (size == 0)
    {
        loclist->locations = NULL;
    }
    else
    {
        loclist->locations = malloc(sizeof(loclist-
>locations[0])*size);
    }
}

static void destroyLocationList(LocationList *loclist)
{
    loclist->num_locations = 0;
    free(loclist->locations);
    loclist->locations = NULL;
}

#ifdef __cplusplus
}
#endif
```

```
#endif
```

3.3.4.3 *Client example*

aostest.c - client implementation of aostest (Array-Of-Structures Test) example

```
/*
 * This file contains client implementation of aostest (Array-Of-
Structures Test) example
 * implemented in C with KIARA framework.
 */

#include <KIARA/kiara.h>
#include <KIARA/kiara_macros.h>
#include <KIARA/kiara_pp_annotation.h>

#include "aostest_types.h"
#include "aostest_kiara_client.h"

#include <stdio.h>
#include <string.h>
#include "c99fmt.h"

/*
 * Declare application structures and client functions of the
application.
 * Following macros are processed by kiara-preprocessor tool and result
 * is output to the aostest_kiara_client.h file.
 */

/* kiara_declare_struct_with_api(type_name, ...)
 *
 * Declares non-trivial structure type type_name, which require custom
behavior via
 * user specified API functions.
 *
 * kiara_struct_array_member(ptr_member_name, size_member_name)
 *
 * Declares member in a structure that represents a C-array composed
from
 * pointer to the array field and integer array size field.
```

```
 *
 * kiara_user_api(api_name, user_function_name)
 *
 * Registers user-defined API function user_function_name with a
predefined KIARA API api_name
 *
 * In our example we need to declare LocationList structure explicitly
because it
 * contains C-array composed from locations pointer and num_locations
integer and
 * because LocationList requires custom allocation and deallocation
functions.
 * Both can't be deduced automatically from the source code.
 */
kiara_declare_struct(LocationList,
    kiara_struct_array_member(locations, num_locations))


/* kiara_declare_func(func_name, ...)
 *
 * Declares remote function that can be called by the client.
 * All arguments after the function name are argument types and names
 * to the function.
 *
 * Usually types used as arguments do not need to be explicitly
declared.
 * Only when types require custom handling an explicit declaration is
required.
 */
kiara_declare_func(AOSTest_SetLocations, const LocationList *
locations)
kiara_declare_func(AOSTest_GetLocations, LocationList * result_value
locations)


/*
 * KIARA context and connection variables.
 *
 * KIARA_Context is used for all KIARA operations issued from the
single thread.
 * Each separate thread require a separate KIARA_Context instance.
 *
 * KIARA_Connection is a handle to the remote endpoint
```

```
 * over which remote calls are performed.
 */
KIARA_Context *ctx;
KIARA_Connection *conn;

/*
 * set_locations and get_locations are handles to the function objects
 * that perform remote call.
 * They are dynamically generated by the KIARA_GENERATE_CLIENT_FUNC
macro.
 */
KIARA_FUNC_OBJ(AOSTest_SetLocations) set_locations;
KIARA_FUNC_OBJ(AOSTest_GetLocations) get_locations;

/*
 * Initialization of the connection
 */
void initConn(const char *url)
{
    /* Create new context */

    ctx = kiaraNewContext();

    /* Open connection to the service */

    printf("Opening connection to %s...\n", url);
    conn = kiaraOpenConnection(ctx, url);

    if (!conn)
    {
        fprintf(stderr, "Error: Could not open connection : %s\n",
kiaraGetContextError(ctx));
        exit(1);
    }

    /*
     * KIARA_GENERATE_CLIENT_FUNC(connection, idl_method_name,
func_type_name, mapping)
     *
```

```
     * Generates function that will perform a remote call.
     *
     * connection       - opened and valid KIARA_Connection handle.
     * idl_method_name  - name of the remote service method specified
in the IDL.
     * func_type_name   - name of the function object type declared
     *                    with the KIARA_FUNC_OBJ(func_type_name)
macro.
     * mapping          - mapping of the IDL types to the application
types.
     *                    By default 1:1 mapping by names and types is
used.
     *                    Note: mapping is not implemented yet.
     *
     * Note: The IDL of the server application is embedded in
aostest_server.c.
     */


    set_locations = KIARA_GENERATE_CLIENT_FUNC(conn,
"aostest.setLocations", AOSTest_SetLocations, "");
    if (!set_locations)
        fprintf(stderr, "Error: code generation failed: %s\n",
kiaraGetConnectionError(conn));


    get_locations = KIARA_GENERATE_CLIENT_FUNC(conn,
"aostest.getLocations", AOSTest_GetLocations, "");
    if (!get_locations)
        fprintf(stderr, "Error: code generation failed: %s\n",
kiaraGetConnectionError(conn));
}


/*
 * Close connection and finalize KIARA framework
 */
void finalizeConn()
{
    kiaraCloseConnection(conn);
    kiaraFreeContext(ctx);
    kiaraFinalize();
}
```

```
int main(int argc, char **argv)
{
    const char *url = NULL;
    int errorCode;

    /* Initialize KIARA */
    kiaraInit(&argc, argv);

    if (argc > 1)
        url = argv[1];
    else
        url = "http://localhost:8080/service";

    /* Initialize connection and generate functions */
    initConn(url);

    /* Call remote functions */

    /* Send 10 locations to the server, where they will be stored */
    {
        size_t num, i;
        LocationList loclist;
        num = 10;
        initLocationList(&loclist, num);
        for (i = 0; i < num; ++i)
        {
            loclist.locations[i].position.x = i;
            loclist.locations[i].position.y = i;
            loclist.locations[i].position.z = i;

            loclist.locations[i].rotation.r = 0.707107f;
            loclist.locations[i].rotation.v.x = 0.0f;
            loclist.locations[i].rotation.v.y = 0.0f;
            loclist.locations[i].rotation.v.z = 0.70710701f;
        }

        /*
         * KIARA_CALL(funcobj, ...)
```

```
         *
         * Will call remote function via function object generated by
KIARA_GENERATE_CLIENT_FUNC macro.
         * All arguments after function objects are input/output
arguments to the remote function.
         * KIARA_CALL returns integer value of type KIARA_Result that
represent an error code.
         */


        errorCode = KIARA_CALL(set_locations, &loclist);
        if (errorCode != KIARA_SUCCESS)
            fprintf(stderr, "Error: call failed: %s\n",
kiaraGetConnectionError(conn));
        else
            printf("aostest.setLocations: DONE\n");
        destroyLocationList(&loclist);
    }

    /* Receive locations stored on the server, and print them */
    {
        size_t num, i;
        LocationList loclist;

        initLocationList(&loclist, 0);

        errorCode = KIARA_CALL(get_locations, &loclist);
        if (errorCode != KIARA_SUCCESS)
            fprintf(stderr, "Error: call failed: %s\n",
kiaraGetConnectionError(conn));
        else
        {
            printf("aostest.getLocations: LocationList {\n");
            printf("  locations: [\n");
            for (i = 0; i < loclist.num_locations; ++i)
            {
                printf("    position %f %f %f rotation %f %f %f %f\n",
                    loclist.locations[i].position.x,
                    loclist.locations[i].position.y,
                    loclist.locations[i].position.z,
                    loclist.locations[i].rotation.r,
```

```
                    loclist.locations[i].rotation.v.x,
                    loclist.locations[i].rotation.v.y,
                    loclist.locations[i].rotation.v.z);
            }
            printf("  ]\n");
            printf("}\n");
        }

        destroyLocationList(&loclist);
    }

    /* Finalize */

    finalizeConn();

    return 0;
}
```

aostest_kiara_client.h - code generated by kiara-preprocessor from aostest.c

```
#ifndef KIARA_PP_0A0ZNAOU2TTQRV0FVWC5_H
#define KIARA_PP_0A0ZNAOU2TTQRV0FVWC5_H

#include <KIARA/kiara.h>
#include <KIARA/kiara_macros.h>

/* This file was generated by kiara-preprocessor tool */

#include "aostest_types.h"

KIARA_DECL_STRUCT(Vec3f,
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, x)
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, y)
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, z)
)
KIARA_DECL_STRUCT(Quatf,
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, r)
  KIARA_STRUCT_MEMBER(Vec3f, v)
)
KIARA_DECL_STRUCT(Location,
```

```
  KIARA_STRUCT_MEMBER(Vec3f, position)
  KIARA_STRUCT_MEMBER(Quatf, rotation)
)
KIARA_DECL_PTR(Location_ptr, Location)
KIARA_DECL_STRUCT(LocationList,
  KIARA_STRUCT_ARRAY_MEMBER(Location_ptr, locations, KIARA_INT,
num_locations)
)
KIARA_DECL_CONST_PTR(const_LocationList_ptr, LocationList)
KIARA_DECL_FUNC(AOSTest_SetLocations,
  KIARA_FUNC_ARG(const_LocationList_ptr, locations)
)
KIARA_DECL_PTR(LocationList_ptr, LocationList)
KIARA_DECL_FUNC(AOSTest_GetLocations,
  KIARA_FUNC_RESULT(LocationList_ptr, locations)
)

#endif
```

3.3.4.4 *Server example*

aostest_server.c - KIARA server implementation of aostest (Array-Of-Structures Test) example

```
/*
 * This file contains server implementation of aostest (Array-Of-
Structures Test) example
 * implemented in C with KIARA framework.
 */

#include <KIARA/kiara.h>
#include <KIARA/kiara_macros.h>
#include <KIARA/kiara_pp_annotation.h>

#include "aostest_types.h"
#include "aostest_kiara_server.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "c99fmt.h"

/*
```

```
 * Declare application structures and service functions of the
application.
 * Following macros are processed by kiara-preprocessor tool and result
 * is output to the aostest_kiara_server.h file.
 */


/* kiara_declare_struct_with_api(type_name, ...)
 *
 * Declares non-trivial structure type type_name, which require custom
behavior via
 * user specified API functions.
 *
 * kiara_struct_array_member(ptr_member_name, size_member_name)
 *
 * Declares member in a structure that represents a C-array composed
from
 * pointer to the array field and integer array size field.
 *
 * kiara_user_api(api_name, user_function_name)
 *
 * Registers user-defined API function user_function_name with a
predefined KIARA API api_name
 *
 * In our example we need to declare LocationList structure explicitly
because it
 * contains C-array composed from locations pointer and num_locations
integer and
 * because LocationList requires custom allocation and deallocation
functions.
 * Both can't be deduced automatically from the source code.
 */
kiara_declare_struct(LocationList,
    kiara_struct_array_member(locations, num_locations))


/* kiara_declare_service(service_name, ...)
 *
 * Declares service function type that can be called by the client.
 * All arguments after the function name are argument types and names
 * to the function.
 *
```

```
 * Usually types used as arguments do not need to be explicitly
declared.
 * Only when types require custom handling an explicit declaration is
required.
 */
kiara_declare_service(AOSTest_GetLocationsImpl, LocationList *
result_value locations)
kiara_declare_service(AOSTest_SetLocationsImpl, const LocationList *
locations)


/** Service implementation */


/* In objectLocations list are stored locations sent by the client */
LocationList objectLocations = {0, NULL};


void copyLocationList(LocationList *dest, const LocationList *src)
{
    if (dest->num_locations != src->num_locations)
    {
        destroyLocationList(dest);
        initLocationList(dest, src->num_locations);
    }
    memcpy(dest->locations, src->locations, sizeof(src->locations[0]) *
src->num_locations);
}


/* Receive location list sent by the client and store it to the
objectLocations variable */
KIARA_Result aostest_set_locations_impl(KIARA_ServiceFuncObj
*kiara_funcobj, const LocationList *locations)
{
    size_t i;
    size_t num = locations->num_locations;
    for (i = 0; i < num; ++i)
    {

printf("Location.position %f %f %f\nLocation.rotation %f %f %f %f\n",
                locations->locations[i].position.x,
                locations->locations[i].position.y,
                locations->locations[i].position.z,
                locations->locations[i].rotation.r,
```

```
                locations->locations[i].rotation.v.x,
                locations->locations[i].rotation.v.y,
                locations->locations[i].rotation.v.z);
    }

    copyLocationList(&objectLocations, locations);

    return KIARA_SUCCESS;
}

/* Return location list stored in the objectLocations variable back to
the client */
KIARA_Result aostest_get_locations_impl(KIARA_ServiceFuncObj
*kiara_funcobj, LocationList *locations)
{
    copyLocationList(locations, &objectLocations);

    return KIARA_SUCCESS;
}

int main(int argc, char **argv)
{
    /*
     * KIARA context and connection variables.
     *
     * KIARA_Context is used for all KIARA operations issued from the
single thread.
     * Each separate thread require a separate KIARA_Context instance.
     *
     * KIARA_Service is a handle to the service which provides
implementation
     * of service methods specified in the IDL.
     *
     * KIARA_Server is a handle to the server which can host multiple
services.
     */

    KIARA_Context *ctx;
    KIARA_Service *service;
    KIARA_Server *server;
```

```
    KIARA_Result result;
    const char *port = NULL;
    const char *protocol = NULL;

    /* Initialize KIARA */
    kiaraInit(&argc, argv);

    if (argc > 1)
        port = argv[1];
    else
        port = "8080";

    if (argc > 2)
        protocol = argv[2];
    else
        protocol = "jsonrpc";

    printf("Server port: %s\n", port);
    printf("Protocol: %s\n", protocol);

    /* Create new context */

    ctx = kiaraNewContext();

    /* Create a new service */

    service = kiaraNewService(ctx);

    /* Add IDL to the service */

    result = kiaraLoadServiceIDLFromString(service,
            "KIARA",
            "namespace * aostest "
            "struct Vec3f {"
            " float x, "
            " float y, "
            " float z "
            "} "
```

```
            "struct Quatf {"
            " float r, "
            " Vec3f v  "
            "} "
            "struct Location {"
            " Vec3f position, "
            " Quatf rotation  "
            "} "
            "struct LocationList { "
            " array<Location> locations "
            "} "
            "service aostest { "
            "  void setLocations(LocationList locations); "
            "  LocationList getLocations(); "
            "} ");
    if (result != KIARA_SUCCESS)
    {
        fprintf(stderr, "Error: could not parse IDL: %s: %s\n",
                kiaraGetErrorName(result),
kiaraGetServiceError(service));
        exit(1);
    }

    printf("Register aostest.setLocations ...\n");

    /*
     * KIARA_REGISTER_SERVICE_FUNC(service, idl_method_name,
     *                             service_type_name, mapping,
service_func_impl)
     *
     * Registers service function implementation with a specified IDL
service method.
     *
     * service           - valid KIARA_Service handle.
     * idl_method_name   - name of the remote service method specified
in the IDL.
     * service_type_name - name of the service type declared with the
KIARA_DECL_SERVICE macro.
     * mapping           - mapping of the IDL types to the application
types.
```

```
     *                      By default 1:1 mapping by names and types is
used.
     *                      Note: mapping is not implemented yet.
     * service_func_impl - user function that implements service
method.
     */


    result = KIARA_REGISTER_SERVICE_FUNC(service,
"aostest.setLocations", AOSTest_SetLocationsImpl, "",
aostest_set_locations_impl);
    if (result != KIARA_SUCCESS)
    {
        fprintf(stderr, "Error: registration failed: %s: %s\n",
                kiaraGetErrorName(result),
kiaraGetServiceError(service));
        exit(1);
    }


    printf("Register aostest.getLocations ...\n");


    result = KIARA_REGISTER_SERVICE_FUNC(service,
"aostest.getLocations", AOSTest_GetLocationsImpl, "",
aostest_get_locations_impl);
    if (result != KIARA_SUCCESS)
    {
        fprintf(stderr, "Error: registration failed: %s: %s\n",
                kiaraGetErrorName(result),
kiaraGetServiceError(service));
        exit(1);
    }


    /*
     * Create new server and register service
     */


    server = kiaraNewServer(ctx, "0.0.0.0", atoi(port), "/service");


    kiaraAddService(server, "/rpc/aostest", protocol, service);


    printf("Starting server...\n");
```

```
    /* Run server */

    result = kiaraRunServer(server);
    if (result != KIARA_SUCCESS)
        fprintf(stderr, "Error: could not start server: %s: %s\n",
                kiaraGetErrorName(result),
kiaraGetServerError(server));

    /* Free everything */

    kiaraFreeServer(server);
    kiaraFreeService(service);
    kiaraFreeContext(ctx);
    kiaraFinalize();

    /* Free temporary copy of location list */
    destroyLocationList(&objectLocations);

    return 0;
}
```

aostest_kiara_server.h - code generated by kiara-preprocessor from aostest_server.c

```
#ifndef KIARA_PP_U759IC9SH7PSLPQOSZB7_H
#define KIARA_PP_U759IC9SH7PSLPQOSZB7_H

#include <KIARA/kiara.h>
#include <KIARA/kiara_macros.h>

/* This file was generated by kiara-preprocessor tool */

#include "aostest_types.h"

KIARA_DECL_STRUCT(Vec3f,
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, x)
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, y)
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, z)
)
KIARA_DECL_STRUCT(Quatf,
  KIARA_STRUCT_MEMBER(KIARA_FLOAT, r)
```

```
  KIARA_STRUCT_MEMBER(Vec3f, v)
)
KIARA_DECL_STRUCT(Location,
  KIARA_STRUCT_MEMBER(Vec3f, position)
  KIARA_STRUCT_MEMBER(Quatf, rotation)
)
KIARA_DECL_PTR(Location_ptr, Location)
KIARA_DECL_STRUCT(LocationList,
  KIARA_STRUCT_ARRAY_MEMBER(Location_ptr, locations, KIARA_INT,
num_locations)
)
KIARA_DECL_PTR(LocationList_ptr, LocationList)
KIARA_DECL_SERVICE(AOSTest_GetLocationsImpl,
  KIARA_SERVICE_RESULT(LocationList_ptr, locations)
)
KIARA_DECL_CONST_PTR(const_LocationList_ptr, LocationList)
KIARA_DECL_SERVICE(AOSTest_SetLocationsImpl,
  KIARA_SERVICE_ARG(const_LocationList_ptr, locations)
)


#endif
```

3.3.5 KIARA Interface definition language

Current KIARA IDL is based on Thirft syntax described here: http://thrift.apache.org/docs/idl/. All modifications to the original syntax are documented below. Major extension is the support of annotations based on Web IDL syntax: http://www.w3.org/TR/WebIDL/ and generic types. A number of Thrift features will be removed in a future and replaced by annotations.

3.3.5.1 ***Document***

```
[1]  Document         ::=  Header* Definition*
```

3.3.5.2 ***Header***

```
[2]  Header           ::=  Include | CppInclude | Namespace
```

3.3.5.2.1 *KIARA Include*

```
[3]  Include          ::=  'include' Literal
```

3.3.5.2.2 *C++ Include*

TODO This should be removed or replaced by an annotation.

```

```

```
[4]  CppInclude       ::=  'cpp_include' Literal
```

3.3.5.2.3 *Namespace*

Thrift uses `namespace` for mapping to different scripting languages. This should be performed by annotations. We plan to make namespace similar to CORBA's `module`, so we can annotate all contents of a single IDL file.

```
[5]  Namespace        ::=  ( 'namespace' ( NamespaceScope Identifier ) |
                           ( 'smalltalk.category'
STIdentifier ) |
                           ( 'smalltalk.prefix' Identifier
) )

[6]  NamespaceScope   ::=  '*' | 'cpp' | 'java' | 'py' | 'perl' | 'rb' |
'cocoa' | 'csharp'
```

3.3.5.3 ***Definition***

Added syntax for defining new annotations.

```
[7]  Definition       ::=  Const | Typedef | Enum | Senum | Struct |
Exception | Service | AnnotationDef
```

3.3.5.3.1 *Const*

```
[8]  Const            ::=  'const' FieldType Identifier '=' ConstValue
ListSeparator?
```

3.3.5.3.2 *Typedef*

```
[9]  Typedef          ::=  'typedef' DefinitionType Identifier
```

3.3.5.3.3 *Enum*

```
[10] Enum             ::=  'enum' Identifier '{' (Identifier ('='
IntConstant)? ListSeparator?)* '}'
```

3.3.5.3.4 *Senum*

```
[11] Senum            ::=  'senum' Identifier '{' (Literal
ListSeparator?)* '}'
```

3.3.5.3.5 *Struct*

Added syntax for annotating structs.

```
[12] Struct           ::=  AnnotationList? 'struct' Identifier
'xsd_all'? '{' Field* '}'
```

3.3.5.3.6 *Exception*

Added syntax for annotating exceptions.

```
[13] Exception       ::=  AnnotationList? 'exception' Identifier '{'
Field* '}'
```

3.3.5.3.7 *Service*

Added syntax for annotating services.

```
[14] Service         ::=  AnnotationList? 'service' Identifier (
'extends' Identifier )? '{' Function* '}'
```

3.3.5.4 ***Field***

Added syntax for annotating fields.

```
[15] Field           ::=  FieldID? AnnotationList? FieldReq? FieldType
Identifier ('= ConstValue)? XsdFieldOptions ListSeparator?
```

3.3.5.4.1 *Field ID*

```
[16] FieldID         ::=  IntConstant ':'
```

3.3.5.4.2 *Field Requiredness*

```
[17] FieldReq        ::=  'required' | 'optional'
```

3.3.5.4.3 *XSD Options*

TODO This should be removed or replaced by an annotation.

```
[18] XsdFieldOptions ::=  'xsd_optional'? 'xsd_nillable'? XsdAttrs?


[19] XsdAttrs        ::=  'xsd_attrs' '{' Field* '}'
```

3.3.5.5 ***Functions***

`oneway` attribute was removed from the original Thrift syntax. Use `Oneway` annotation and void return type for representing functions that never return a value. Support for annotating function and its return type was added.

```
[20] Function        ::=  AnnotationList? FunctionType AnnotationList?
Identifier '(' Field* ')' Throws? ListSeparator?


[21] FunctionType    ::=  FieldType | 'void'


[22] Throws          ::=  'throws' '(' Field* ')'
```

The first annotation list annotates function as a whole, e.g.:

```
[Bar(22), Baz] void foobar(i32 i, float f);
```

The annotation list after function return type annotates only return type, e.g.:

```
void [Bar(22), Baz] foobar(i32 i, float f);
```

Each function parameter can be annotated separately, e.g.:

```
void foobar([Bar(22)] i32 i, [Baz] float f);
```

Finally, all these cases can be combined together:

```
[Bar(22), Baz] void [Foo(22), Bar] foobar([Bar(22)] i32 i, [Baz] float
f);
```

3.3.5.6 *Types*

Thrift's `bool` type was renamed to `boolean`, `byte` type was renamed to `i8`. Unsigned integer types `u8`, `u16`, `u32`, and `u64` were added. Instead of using predefined `list`, `map` and `set` container types KIARA syntax allows to use arbitrary C++/CORBA like generic types: `list< i32>`, `array< i32, 2>`, `array< i32, array< i32, 4> >`.

```
[23] FieldType       ::=  Identifier | BaseType | GenericType

[24] DefinitionType  ::=  BaseType | GenericType

[25] BaseType        ::=  'boolean' | 'i8' | 'i16' | 'i32' | 'i64' |
'u8' | 'u16' | 'u32' | 'u64' | 'double' | 'string' | 'binary' | 'slist'

[26] GenericType     ::=  Identifier '<' GenericTypeArg (','
GenericTypeArg)* '>'

[27] GenericTypeArg  ::=  Identifier | BaseType | GenericType |
IntConstant | DoubleConstant | Literal | ConstList | ConstMap
```

3.3.5.7 *Constant Values*

```
[28] ConstValue      ::=  IntConstant | DoubleConstant | Literal |
Identifier | ConstList | ConstMap

[29] IntConstant     ::=  ('+' | '-')? Digit+

[30] DoubleConstant  ::=  ('+' | '-')? Digit* ('.' Digit+)? ( ('E' |
'e') IntConstant )?

[31] ConstList       ::=  '[' (ConstValue ListSeparator?)* ']'

[32] ConstMap        ::=  '{' (ConstValue ':' ConstValue
ListSeparator?)* '}'
```

3.3.5.8 ***Basic Definitions***

3.3.5.8.1 *Literal*

```
[33] Literal          ::=  ('"' [^"]* '"') | ("'" [^']* "'")
```

3.3.5.8.2 *Identifier*

```
[34] Identifier       ::=  ( Letter | '_' ) ( Letter | Digit | '.' | '_'
)*

[35] STIdentifier     ::=  ( Letter | '_' ) ( Letter | Digit | '.' | '_'
| '-' )*
```

3.3.5.8.3 *List separator*

```
[36] ListSeparator    ::=  ',' | ';'
```

3.3.5.8.4 *Letters and Digits*

```
[37] Letter           ::=  ['A'-'Z'] | ['a'-'z']

[38] Digit            ::=  ['0'-'9']
```

3.3.5.9 ***Annotations***

This part is specific to KIARA IDL.

```
[39] AnnotationList   ::= '[' Annotation (',' Annotation)* ']'

[40] Annotation       ::= Identifier AnnotationArgs?

[41] AnnotationArgs   ::= '(' Identifier ('=' ConstValue)? (','
Identifier ('=' ConstValue)? )* ')'
```

User can define new annotations using following syntax:

```
[42] AnnotationDef   ::= AnnotationList? 'annotation' Identifier '{'
Field* '}'
```

3.3.6 Interface Examples

```
namespace * calc


exception DivisionByZero {
}
```

```
// service annotation
[HTTPPort(8080)]
service Calculator {

  // function annotation
  [Oneway]
  void ping()

  float add(float a, float b)
  float sub(float a, float b)
  float div(float a, float b) throws (DivisionByZero err)
  float mul(float a, float b)

  array<i32> addArray(array<i32> a, array<i32> b)
}
```

4 Middleware - RPC over DDS - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

4.1 Introduction

FI-WARE Middleware GE (code named KIARA) is a new middleware based on the Data Distribution Service (DDS) specifications, an OMG Standard defining the API and Protocol for high performance publish-subscribe middleware, and RPC over DDS, an Remote Procedure Call framework using DDS as the transport and based on the ongoing OMG RPC for DDS standard. A quick DDS introduction is provided here

In contrast to other GEs, the FI-WARE Middleware GE is not a standalone service running in the network, but a set of compile-/runtime tools and a communication library to be delivered with the application.

In this release of the middleware for FI-WARE (R3.3), we are providing the basic assets, DDS and RPC for DDS, and other modules that are the building blocks of the KIARA Middleware

RPC for DDS was updated to include several planned features for KIARA, such as asynchronous calls and a high performance dispatching agent.

In this release RPC over DDS is fully compatible with other building block called RPC over REST. Using the same IDL and API you can call Remote Procedure Calls using DDS as the underlying transport, or call REST services. This feature is one of the key requirements of FI-WARE Middleware.

4.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Advanced Middleware GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

4.1.2 Software Options

Two different implementations of DDS can be selected:

1. RTI DDS 5.0 or later (Free, Open Community Source License)
1. OpenDDS 3.4.1 or later (Free, Open Source License)

We recommend RTI DDS because its performance and ease of use.

For the remote procedure calls framework use

- eProsima RPC for DDS 0.3.0 (free and Open Source License).

4.2 User Guide

These products are for programmers, who will invoke the APIs programmatically and there is no user interface as such.

See the programmers guide section to browse the available documentation.

4.3 Programmers Guide

All the available documentation for this release is published online.

4.3.1 DDS

4.3.1.1 *RTI DDS*

RTI DDS documentation is published online here, including User Manual, API Reference, Best Practices, Examples…

A "community portal" is available, with Forums, Knowledge base and more.

4.3.1.2 *OpenDDS*

Open DDS documentation is published online here, including a Developer's Guide, API Reference, examples…

The OpenDDS Web contains also articles, faqs, developer tools, etc.

4.3.2 eProsima RPC for DDS

RPC for DDS is published online in the Product Page, including a User Manual and the API Reference:

- User Manual (PDF)
- API Reference - Doxygen (HTML) (PDF)

If you need assistance, please contact eProsima Support

5 Middleware - RPC over REST - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

5.1 Introduction

One of the main FI-WARE middleware requirements is to offer backwards compatibility with Web Services, specifically RESTful Web Services.

RPC over REST enable the creation and the invocation of RESTful Web Services through the common API used in RPC over DDS and the future RPC over TCP, both part of the FI-WARE middleware suite.

This product ease the integration or migration of existing web services with the FI-WARE middleware.

RPC over REST supports WADL (Web Application Definition Language) as the IDL to define RESTful Web Services.

5.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Advanced Middleware GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

5.2 User guide

These products are for programmers, who will invoke the APIs programmatically and there is no user interface as such.

See the programmers guide section to browse the available documentation.

5.3 Programmers guide

RPC over TCP doc are published online in eProsima Website, including a User Manual and the API Reference:

- User Manual (PDF)
- API Reference - Doxygen (HTML) (PDF)

If you need assistance, please contact eProsima Support

6 Middleware - Fast Buffers - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

6.1 Introduction

Fast Buffers is an open source serialization engine optimized for performance, beating alternatives such as Apache Thrift and Google Protocol Buffers in both Simple and Complex Structures.

Fast Buffers generates serialization code for your structured data from its definition in an Interface Description Language (IDL).

Fast Buffers is the high performance serialization mechanism of the KIARA middleware suite

6.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Advanced Middleware GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

6.2 User guide

These products are for programmers, who will invoke the APIs programmatically and there is no user interface as such.

See the programmers guide section to browse the available documentation.

6.3 Programmers guide

Fast Buffers is published online in the Product Page, including a User Manual and the API Reference:

- User Manual (PDF)
- API Reference - Doxygen (HTML) (PDF)

If you need assistance, please contact eProsima Support

7 2D-UI - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

7.1 Introduction

This document describes how to use the Input API and develop a web application or web page which make use of the Input API. In addition document describes how to start using and creating Polymer Web Components.

The Input API is a Javascript library which based on common Javascript libraries such as JQuery.js and signals.js. Polymer library uses latest web technologies and enables creating custom HTML elements. For both the basic Javascript, HTML and CSS knowledge is needed. Understanding Polymer concept get familiar with the basic information about Web Components http://www.w3.org/TR/components-intro/.

7.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the 2D-UI GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description

7.2 User guide

This section does not apply. All components related to 2D-UI are for programmers and are used programmatically.

7.3 Programmers guide

7.3.1 InputAPI

How to download and install the Input API is described in the 2D-UI - Installation and Administration Guide

To start using the libraries have a look at the **`input.html`** file in the **`test`** folder. This file shows how to use the Input API and the IInputPlugin for creating new input plugins. InputAPI uses jQuery, jQuery plugins, Signals and Classy JavaScript libraries.

At first create a html page and include needed JavaScript files to the <head> section of your newly created file. Remember that all references to plugin files must be placed after the main InputAPI.js reference which contains the definition of IInputPlugin.

If you use the structure of the cloned git repository, place the file under **`test`** folder and leave script paths as they are. If you move the file to e.g. your own web server change the paths accordingly.

```
<!-- jQuery -->
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>

<!-- Dependencies: jQuery plugins -->
<script src="../lib/jquery.mousewheel.js"></script>
<script src="../lib/jquery.hotkeys.js"></script>
<script src="../lib/jgestures.min.js"></script>  <!-- touch events -->
```

```
<script src="../lib/signals.js"></script>        <!-- signaling -->


<!-- Dependencies: Libraries -->
<script src="../lib/classy.js"></script>                <!-- classes
with proper inheritance -->


<!-- Input API-->
<script src="../src/InputAPI.js"></script>
```

Instantiate the InputAPI and give it a container element. The container can be empty in the initialization. All signals can be registered manually afterwards. For keyboard signals given elements must contain tabIndex property, otherwise signals will be bound to the document and keyboard signals won´t work properly. The container library uses jQuery to find corresponding HTML elements in the document. In this example we have created a div element which ID is input-console. Place all the described sections below inside a script tag: <script type="text/javascript"></script>.

```
var inputAPI = new InputAPI({
    //Give container for event registering. Not mandatory. Register can
be done manually.
    container: "#input-console"
});
```

At this stage the InputAPI has bound all generic mouse and keyboard signals which are included in the InputAPI itself.
Next start listening those signals by adding a listener.

```
inputAPI.mouseEvent.add(onMouseEvent);
inputAPI.keyEvent.add(onKeyEvent);
```

Here is a complete list of supported keyboard and mouse events. Keyboard:

- keyEvent ("press" | "release")
- keyPress
- keyRelease

Mouse:

- mouseEvent ("move" | "press" | "release" | "wheel")
- mouseMove
- mousePress
- mouseRelease
- mouseClick
- mouseWheel

Create corresponding listener functions. Read the data and use it for your purposes. In this example the data is just pushed to the given container.

```
function onKeyEvent(event)
{
    $('#input-console').prepend("<b style=\"color: blue;\">Key:
"+event.key +" - "+event.type+"</b><br>");
}

function onMouseEvent(event)
{
    $('#input-console').prepend("<b style=\"color: red;\">Mouse:
"+event.type+":</b> X="+event.x+", Y="+event.y+"<br>");
}
```

All mouse signals contain a mouse/event object which keeps its state and provides the following data to use:

```
    //Event type: move, press, release, wheel
    type : "",
    // Absolute position
    type : "",
    // Absolute position
    x : null,
    y : null,
    // Relative position
    relativeX : 0,
    relativeY : 0,
    // Wheel delta
    relativeZ : 0,
    // Button states
    rightDown  : false,
    leftDown   : false,
    middleDown : false,
    // HTML element id that the mouse event occurred on
    targetId : "",
    // HTML node name eg. 'canvas' and 'div'. Useful for detected
    // when on 'canvas' element aka the mouse event occurred on the
    // 3D scene canvas and not on top of a UI widget.
    targetNodeName : "",
    // Original jQuery mouse event
```

```
    originalEvent : null
```

All keyboard signals contain a keyboard/event object which keeps its state and provides following data to use:

```
    // Event type: press, release
    type : "",
    // Event key code
    keyCode : 0,
    // Event key as string
    key : "",
    // If this is a repeat. Meaning the key was already in the pressed
state.
    repeat : false,
    // Currently held down keys: maps key as string to 'true' boolean
    // Check with inputApi.keyboard.pressed["w"] or
keyEvent.pressed["f"]
    pressed : {},
    // HTML element id that the mouse event occurred on
    targetId : "",
    // HTML node name eg. 'canvas' and 'div'. Useful for detected
    // when on 'body' element aka the mouse event occurred on the "3D
scene" and not on top of another input UI widget.
    targetNodeName : "",
    // Original jQuery mouse event
    originalEvent : null
```

For a complete example please look at the input.html file inside the test folder of the cloned repository. The plugin system is opened and described below.

7.3.1.1 *IInputPlugin*

IInputPlugin provides plugin system for InputAPI. IInputPlugin is included in InputAPI. To create a plugin write new js -file which extends the IInputPlugin. Written plugins are automatically registered as a part of InputAPI. As an example you can find 2 example plugins InputGamepadPlugin.js and InputTouchPlugin.js in the src -folder of the cloned repository. For this guide we will take a deeper look at the TouchInputPlugin.js -file. As default all plugin signals are bound to the given container in the InputAPI. If the container is not present registering can be done manually by giving desired container.

IInputPlugin -class is displayed below:

```
var IInputPlugin = Class.$extend(
{
    __init__ : function(name)
    {
```

```
        if (name === undefined)
        {
            console.error("[IInputPlugin]: Constructor called without a
plugin name!");
            name = "Unknown";
        }
        this.name = name;
        this.running = false;
    },

    __classvars__ :
    {
        register : function()
        {
            var plugin = new this();
            InputAPI.registerPlugin(plugin);
        }
    },

    _start : function(container)
    {
        this.start(container);
        this.running = true;
    },

    start : function()
    {
        console.log("[IInputPlugin]: Plugin '" + name + "' has not
implemented start()");
    },

    _stop : function()
    {
        this.stop();
        this.running = false;
    },

    stop : function()
    {
```

```
        console.log("[IInputPlugin]: Plugin '" + name + "' has not
implemented stop()");
    },

    reset : function()
    {
    }
});
```

To create own IInputPlugin please look at the given examples in the src folder of the cloned repository.

To use created plugins first include the created plugin js -file to the head of your html -file under InputAPI -sript reference. As an example we will use the InputTouchPlugin.js found in the src -folder.

```
<!-- Touch input -->
<script src="../src/InputTouchPlugin.js"></script>
```

Get hold of automatically created plugin by its name and start listening the signals provided by the plugin.

```
var touch = inputAPI.getPlugin("Touch");
if (touch)
{
    //If you want to register the events for a different container that
the inputAPI may already have do it here.
    //touch.registerTouchEvents("#my-div-element");
    //Add a listener to touchEvent signal
    touch.touchEvent.add(onTouchEvent);
}

function onTouchEvent (obj, event)
{
    $('#input-console').prepend("<b style=\"color: green;\">Touch:
"+obj.type+"</b>: X="+obj.startx+", Y="+obj.starty+",
RelativeX="+obj.relativeX+", RelativeY="+obj.relativeY+",
Moved="+obj.moved+"<br>");
}
```

Here is the complete list of InputTouchPlugin.js supported events:

- touchEvent ("tapone" | "taptwo" | "tapthree" | "tapfour" | "pinch" | "pinchopen" | "pinchclose" | "rotate" | "rotatecw" | "rotateccw" | "swipeone" | "swipetwo" | "swipethree" | "swipefour" | "shake")

- tapOne
- tapTwo
- tapThree
- tapFour
- swipeMove
- swipeOne
- swipeTwo
- swipeThree
- swipeFour
- pinch
- pinchOpen
- pinchClose
- rotate
- rotateCW
- rotateCCW
- shake

To find out the data touch/event -object please look at the TouchInputPlugin.js -file.

7.3.1.2 *InputAbstraction*

InputAPI contains implementation where user can register/unregister and update input contexts with parametrized conditions. The implementation is built up as InputState -class. User can register 0-n InputState objects to InputAPI. Once registering succees a signal is given as return value so user can hook the signal to match own purposes. For example if user states pressing 'w' and 'f' together means 'forward' and the conditions are true InputAPI fires a signal connected to the InputState. Now user can decide what to do when the signal has fired.

InputState -class is described below:

```
/**
    Provides input state for abstraction
    @class InputState
    @constructor
*/
var InputState = Class.$extend(
{

    __init__ : function(params)
    {

        this.name = params.name || ""; //Name of the input state.
Indexing property. Has to be unique within InputAPI context.
```

```
        this.keyBindings = params.keyBindings || null; //Keybindings as
string array.

        this.mouseDown = params.mouseDown || null; //Mouse conditions
LEFT_DOWN, MIDDLE_DOWN, RIGHT_DOWN.

        this.timeslot = params.timeslot || 0; //Time slot within the
given conditions must be true. 0 - 5000 milliseconds, where 0 means no
time slot.

        this.priority = params.priority || 100; //Priority 0 - 100
representing percentage of importance.

        this.multiplier = params.multiplier || 0; //Multiplier 0 - 5.
How many times either mouse or keyboard conditions must be true within
given time slot.

        this.actionSignal = null; //The action signal fired when all
conditions are true.

    },


    __classvars__ :
    {
        Mouse :
        {
            LEFT_DOWN : 1,
            RIGHT_DOWN : 2,
            MIDDLE_DOWN : 3
        }


    },


    reset : function()
    {
        this.actionSignal.removeAll();
    },


    //Set name of the input state. Name can represent what the action
should do e.g. Move forward.
    setName : function(paramName)
    {
        if (paramName)
            this.name = paramName;
    },


    //Set keybindings in string array e.g. [w] representing which keys
must be pressed.
```

```
    setKeyBindings : function(paramKeyBindings)
    {
        if (paramKeyBindings)
            this.keyBindings = paramKeyBindings;
    },

    //Set the pressed mouse button value.
    setMouseDown : function(paramMouseDown)
    {
        if (paramMouseDown)
            this.mouseDown = paramMouseDown;
    },

    //Set time slot within the given mouse, keyboard and multiplier
conditions must be true. 0 - 5000 milliseconds, where 0 means no time
slot.
    setTimeslot : function(paramTimeslot)
    {
        var tsval = parseInt(paramTimeslot);
        if (isNaN(tsval))
        {
            console.log("[InputState] Time slot update failed. Value
must be between 0 and 5000 milliseconds.");
            return false;
        }

        if (tsval >= 0 && tsval <= 5000)
        {
            this.timeslot = tsval;
        }
        else
        {
            console.log("[InputState] Time slot update failed. Value
must be between 0 and 5000 milliseconds.");
            return false;
        }
    },

    //Set priority for this inputState. If priority is 100 the state is
handled first. If the priority is 0 the state is handled last.
```

```
    setPriority : function(paramPriority)
    {
        var prval = parseInt(paramPriority);
        if (isNaN(prval))
        {
            console.log("[InputState] Priority update failed. Value
must be between 0 and 100.");
            return false;
        }
        if (prval >= 0 && prval <= 100)
        {
            this.priority = prval;
        }
        else
        {
            console.log("[InputState] Priority update failed. Value
must be between 0 and 100.");
            return false;
        }
    },


    //Set multiplier from 0 to 5. How many times either mouse or
keyboard conditions must be true within given time slot. E.g. if you
give multiplier 2, timeslot 500 and mouse
    //condition says press LEFT_DOWN, the event is fired when user
presses mouse left twice within 500 milliseconds.
    setMultiplier : function(paramMultiplier)
    {


        var mpval = parseInt(paramMultiplier);
        if (isNaN(mpval))
        {
            console.log("[InputState] Multiplier update failed. Value
must be between 0 and 5.");
            return false;
        }


        if (mpval >= 0 && mpval <= 5)
        {
            if (this.timeslot == 0)
```

```
            {
                console.log("[InputState] Time slot cannot be 0 if
multiplier is set");
                return false;
            }

            this.multiplier = mpval;
        }
        else
        {
            console.log("[InputState] Multiplier update failed. Value
must be between 0 and 5.");
            return false;
        }
    }
});
```

7.3.1.2.1 *Creating an InputState*

To start using the InputState have a look at the **`InputState.html`** file in the **`test`** folder. This file shows how to use the input abstraction with the InputState -class. Please follow the guide of InputAPI how to create an html page. Then in addition include the needed InputState.js -file.

```
<!-- InputState -->
<script src="../src/InputState.js"></script>
```

At first instantiate InputAPI as guided in the InputAPI -section and create an InputState:

```
var inputAPI = new InputAPI({
     //Give container for event registering. Not mandatory. Register
can be done manually.
     container: "#my-container"
});

var inputState = new InputState ({
       name : "Name of my input state",
       keyBindings : ['s','w'], //Keybindings as string array
       mouseDown : InputState.Mouse.LEFT_DOWN, //Mouse conditions
LEFT_DOWN, MIDDLE_DOWN, RIGHT_DOWN
       timeslot : 0, //Time slot within the given conditions must be
true. 0 - 5000 milliseconds, where 0 means no time slot
```

```
        priority : 100, //Priority 0 - 100 representing percentage of
importance
        multiplier : 0 //Multiplier 0 - 5. How many times the conditions
must be true within given time slot
});
```

Create an event handler and hook the output e.g. to some div -element on your html -page:

```
function onInputSignal(event)
{
     $('#my-container').prepend("InputState "+inputState.name+"
fired!");
}
```

Register the created InputState. If InputState is well formed InputAPI returns a signal, otherwise false:

```
//Register InputState
var inputStateSignal = inputAPI.registerInputState(inputState);
```

Hook the signal:

```
if (inputStateSignal)
{
     inputStateSignal.add(onInputSignal);
}
else
{
     console.log("InputState has invalid values.");
}
```

7.3.1.2.2 *Update existing InputState*

Change the values of your already registered InputState and update the state:

```
inputAPI.updateInputState(inputState);
```

7.3.2 Polymer Web Component

How to download and install the Polymer Web Components is described in the 2D-UI - Installation and Administration Guide

The downloaded ZIP archive contains 3 examples of a chat web component created with Polymer library version 0.1.1.

- index.html
- dynamic.html

- vulcanized.html

Inside browser_components folder the Polymer project files are located. You can follow the current version of the Polymer Project at GitHub. The chat folder contains the implemented example of chat web components which has the name polymer-chat.

7.3.2.1 *polymer-chat in index.html*

When the index.html file is opened in a browser you can see a chat application written as a Polymer web component.

To use this polymer-chat web component first reference platform.js and import polymer.js which is included in polymer.html to your page <head> section:

```
    <script src="bower_components/platform/platform.js"></script>
    <link rel="import" href="bower_components/polymer/polymer.html">
```

Then import the Polymer web component created:

```
    <link rel="import" href="chat/polymer-chat.html">
```

If you look inside the imported polymer-chat.html you can find it importing a polymer-collapse.html and referencing a css file. At last put the template tag to your page inside <body> tag:

```
    <polymer-chat></polymer-chat>
```

The source code of example page is as follows:

```
<html>
  <head>
    <script src="bower_components/platform/platform.js"></script>
    <link rel="import" href="bower_components/polymer/polymer.html">
    <link rel="import" href="chat/polymer-chat.html">
  </head>
  <body>
    <polymer-chat></polymer-chat>
  </body>
</html>
```

Inside the index.html source code you can also find how to use the polymer-chat:

```
<script>
  //Simple example to use polymer with javascript
  window.addEventListener('WebComponentsReady', function(e) {
      var pchat = document.querySelector('polymer-chat');
      pchat.clientusername = "User 1";
```

```
      pchat.addUser("test", "1");
      pchat.addUser("test2", "2");


      //Connect to polymer fired event
      pchat.addEventListener('chatmessage', function(e)
      {
          console.log("Event fired in polymer component :
"+e.detail.uname+"-"+e.detail.message);
      });


      pchat.onServerMessage("ChatInfoMessage|Server:Message from
server.");
  });
</script>
```

7.3.2.2 *polymer-chat in vulcanized.html*

This example file is similar to the index.html except it uses vulcanized import file. Vulcanize concatenates a set of Polymer Web Components into one file GitHub. Instead of importing polymer-chat.html we now import polymer-chat-full.html which as vulcanized set of polymer-chat and polymer-collapse.

```
  <link rel="import" href="chat/polymer-chat-full.html">
```

If you look inside the polymer-chat-full.html file you can see all templates, css and scripting as a one file.

7.3.2.3 *polymer-chat in dynamic.html*

One of the biggest issues with Polymer is the lack of ability to load web components dynamically. However all needed functions are already there but the Polymer Project itself has not yet created a convenient way of using them. This is discussed topic inside the project and will be part of the project in the future. In this example we show a way to do it currently.

The difference to other examples is that this has to be installed on a web server. The example sets default paths to http://localhost. When trying remember to change these paths accordingly. Change the paths in dynamic.html and polymer-chat-dynamic.html -files.

In this example we use jQuery to download the component therefore a link to jQuery library is needed:

```
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
```

Next we download the polymer-chat-dynamic.html and create the polymer-chat -element on the fly. The content of downloaded file is put to innerHTML of the newly created component and then added to document body -section.

```
   //Download polymer dynamically and at tag to page.
   $.get( "http://localhost/polymer/chat/polymer-chat-dynamic.html",
function( data ) {
```

```
     var element = document.createElement("polymer-chat");
     element.innerHTML = data;
     $("body").append(element);
   });
```

Next we will wait for WebComponentsReady event and start to hook things up. We also have to wait for the polymer-chat component itself to inform that it is ready. For this we wait for polymerchatready -event which is fired when the polymer-chat -component has been fully initialized and ready to use. You can find this piece of code inside the polymer-chat-dynamic.html -file:

```
ready : function()
  {
    console.log("Polymer Chat is ready!");
    this.fire("polymerchatready");
  }
```

Below is the complete script:

```
   //Wait for WebComponentsReady event
   window.addEventListener('WebComponentsReady', function(e) {
       var pchat = document.querySelector('polymer-chat');
       //Wait for PolymerChat being ready
       pchat.addEventListener('polymerchatready', function(e)
       {
           pchat.clientusername = "User 1";
           pchat.addUser("test", "1");
           pchat.addUser("test2", "2");

           //Connect to polymer fired event
           pchat.addEventListener('chatmessage', function(e)
           {
               console.log("Event fired in polymer component :
"+e.detail.uname+"-"+e.detail.message);

           });

           pchat.onServerMessage("ChatInfoMessage|Server:Message from
server.");

       });
   });
```

Since Polymer is at pre-alpha stage all these examples may break if the Polymer version is being updated.

To start programming Polymer all the needed information is provided here: Polymer Project. Polymer is very heavily under development. The best way to get started is to follow instructions of getting started guide: Getting Started.

8 3D-UI - XML3D - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

8.1 Introduction

This documentation introduces how to use XML3D and accelerated Xflow operators. It describes in detail how application developers can use XML3D to declare 3D scenes in the Web-site source code and how DOM API can be used to create and modify scenes during run-time. It moreover explains how to employ hardware acceleration for Xflow, the data-flow that is used by XML3D for complex computations.

This document will be updated as new features are implemented.

8.1.1.1 *Background and Detail*

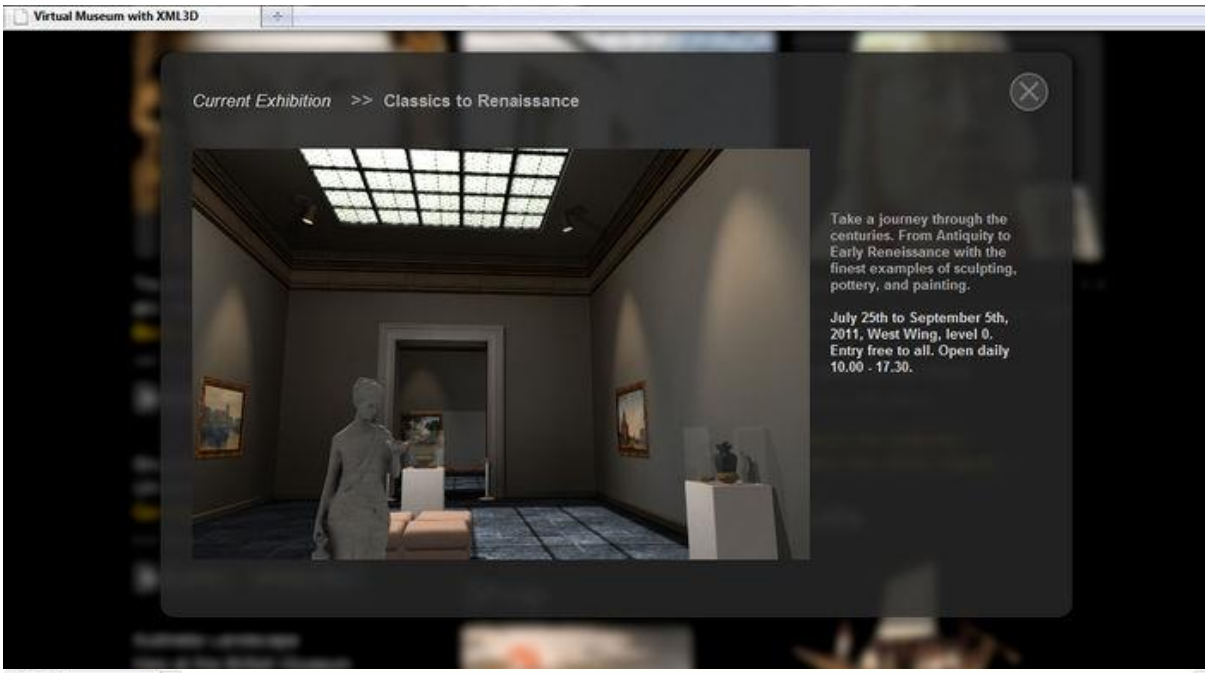
These guides relate to the 3D-UI-XML3D implementation of the 3D-UI Generic Enabler which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more details about this Generic Enabler in the according Architecture and Open Specification documents.

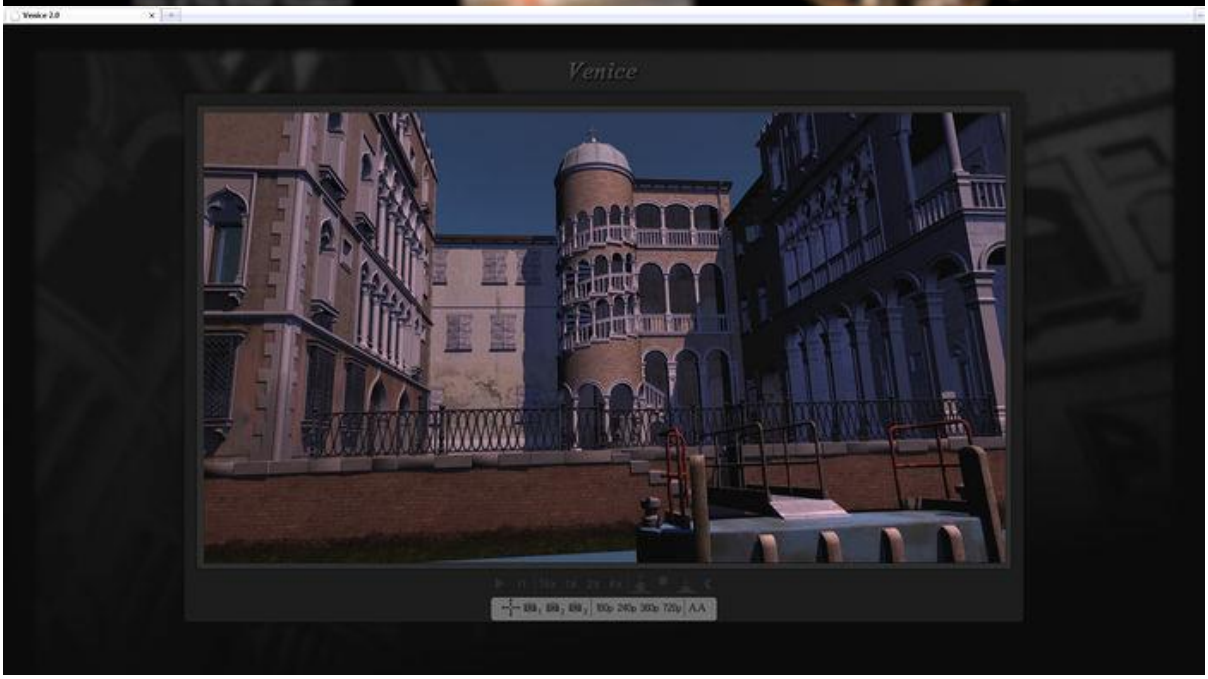
8.2 User guide

Users experience the 3D environment provided in XML3D as soon as they open a web page containing XML3D content.

Depending on the camera controller that is used by the scene, users can navigate through the environment using mouse and keyboard input.

Examine Mode: The camera will stay focused one one point of the scene. Pressing and holding *Left* mouse button will rotate the camera around this point when moving the mouse. The camera will change its distance to the point when moving the mouse while pressing and holding the *Right* mouse button.
Walk Mode: The camera will pretty much behave like known from first-person games. Pressing and holding the *Left* mouse button will rotate the camera around its center point. The user can navigate freely through the scene by using *W, A, S* and *D* keys.





8.3 Programmers guide

There are two ways to create a 3D scene in XML3D: Either you can declare it in an XML-like manner directly in the Web site's source, or create it dynamically with JavaScript coe.

The steps to create a 3D scene are as follows:

- Create a HTML Web-Page
- Link xml3d.js and camera.js as described in the Installation Guide
- Add an xml3d-Element and compose your scene from the nodes given in the Specification
- Upload your Web-page to the server, using for example an FTP client

8.3.1 Adding a mesh in the declarative way

We have already established, that all our 3D content is placed inside an <xml3d> element, so let's start with that:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <script type="text/javascript"
src="http://www.xml3d.org/xml3d/script/xml3d.js"></script>
  <script type="text/javascript"
src="http://www.xml3d.org/xml3d/script/tools/camera.js"></script>
  <title>My very first XML3D teapot</title>
</head>
<body>
    <xml3d xmlns="http://www.xml3d.org/2009/xml3d" >
    </xml3d>
</body>
</html>
```

To add some 3D geometry to our scene, we use the <mesh> element. Just as with <img>, the <mesh> element links an external file containing the 3D geometry data. Here is one of these files for the good ol' teapot: teapot.json

Including the mesh is now fairly simple:

```
<xml3d xmlns="http://www.xml3d.org/2009/xml3d" >
  <mesh src="resource/teapot.json" />
</xml3d>
```

To have the mesh appear nicely in the view of the camera, we have to consider the following:

- the XML3D viewpoint by default is set at the origin of 3D space (x=y=z=0), looking at direction (0,0, -1), so in negative z direction.
- most meshes tend to be places around the origin as well

So consequently: our viewpoint is right next to the surface to the mesh.

Let's change the viewpoint to get some distance to the object. XML3D provides the <view> element to define the viewpoint of the scene:

```
<xml3d xmlns="http://www.xml3d.org/2009/xml3d">
  <view position="0 0 100" />
  <mesh src="resource/teapot.xml#mesh" />
</xml3d>
```

Since our viewpoint looks into the direction (0,0,-1), we move along the z coordinate to get some distance, thus the new position (0,0,100).

Now we start to recognize our teapot, but it's still a bit high up, not quite inside the center. We could now simplt move the viewpoint up a bit again. However, thanks to relativity, there is a second option: Let's move the mesh instead.

In order to move the mesh, we first have to wrap it inside a <group> element. The group element can enclose multiple <mesh> and other <group> elements and transform all of it's content.

To declare the transformation, we simply use CSS 3D Transformations :

```
<xml3d xmlns="http://www.xml3d.org/2009/xml3d">
   <view position="0 0 100" />
   <group  style="transform: translate3d(0px,-20px, 0px)" >
     <mesh src="resource/teapot.xml#mesh" />
   </group>
 </xml3d>
```

The declared transformation moves the teapot downwards, placing it nicely centered in the viewport.

8.3.2 Composition of complex assets from several meshes

Single meshes are very limited when it comes to the declaration of more complex model concepts. For example, one can only assign one single shader to meshes. Also, one may think of models where only parts should be transformable individually by rigid body transformations, like for example a helicopter with moving rotor blades. This could be solved by adding the respective meshes independently in distinct group nodes. However, this potentially adds a lot of <group> and <mesh> nodes to the DOM for basically one single object. For this case, XML3D comes with a more powerful approach: Externally defined *assets.*

Assets are basically a container for a list of <*assetmesh*> elements that define which meshes should be part of the resulting assets. An <assetmesh> can reference an external JSON or XML file just as also done by a <mesh> node, and carry own translations and shaders. <*assetdata*> elements can be used to specify extra sets of data, which may be used by the <assetmesh> via the *includes* attribute. Data that is included by meshes overrides data introduced by the mesh and can therefore be used for individual configuration.

An asset definition could look like this:

```
  <!-- External Asset Definition -->


  <asset id="exampleAsset">
    <assetdata name="config">
      <float3 name="diffuseColor">0.5 0.5 0</float3>
    </assetdata>
    <assetmesh src="mesh1.json" shader="meshdata.xml#shader1"
transform="meshdata.xml#localTransform1"></assetmesh>
    <assetmesh src="mesh2.json" shader="meshdata.xml#shader2"
transform="meshdata.xml#localTransform2" includes="config"></assetmesh>
  </asset>
```

Here, we introduce an asset called *exampleAsset.* It consists of two meshes, each defined in separate JSON files. Shaders and transformations are assigned from a shared definition file called *meshdata.xml.* Mesh2 in addition includes the "configuration" <assetdata> and overwrites the diffuse color which may already be set by *meshdata.xml#shader2.*

Assets that are defined in the above way are not yet added to the scene, and thus not yet rendered. To have them actually appear in the, one has to create an instance using the *<model> element:*

```
<!-- Instance of asset with default parameters -->
<model src="asset.xml#exampleAsset"></model>

<!-- Instance of asset with custom diffuse color -->
<model src="asset.xml#exampleAsset>
  <assetdata name="config">
    <float3 name="diffuseColor">0.7 0.7 1.0</float3>
  </assetdata>
</model>
```

Here, we can notice two things: First, an asset that is referenced by a <model> tag does not necessarily need to be defined within the same document as the model, but can be stored elsewhere, keeping DOM and memory consumption small. Second, also <model> elements can contain <assetdata> nodes. These <assetdata> nodes overwrite the values of the nodes with the same name specified in the <asset> itself. Thus, the code above will render two models in the scene: First, an instance of the asset exactly as given in the definition with a yellow appearance, second an instance which is also an instance of the default asset, but with a individually configured blueish diffuse color. If an asset contains several <assetdata> nodes, or an <assetdata> node contains more than one parameter, one can also operate just on the subset of those which one wants to change.

8.3.3 Dynamic creation of XML3D scenes in JavaScript

XML3D scenes can be created and modified during runtime. This can be done by the standard DOM API of the browser, or by popular frameworks such as jQuery. Basically every tool library that can be used to create dynamic Web-sites can also be employed to modify XML3D scenes.

New elements are created using the respective function of the XML3D namespace:

```
XML3D.createElement(elementName)
```

with *elementName* corresponding to the nodes given in the XML3D Open API Specification

A new mesh contained in a group node can thus be created and added to the scene by the following code:

```
// Create a group and mesh to be added by the scene
var group = XML3D.createElement("group");
var mesh = XML3D.createElement("mesh");
mesh.type= "triangles";
mesh.src = "#meshData";
group.appendChild(mesh);
// Get the XML3D element via jquery and append the new group
$("xml3d").append(group);
```

This code will create the following XML3D tree:

```
<xml3d>
```

```
    <group>
     <mesh type="triangles" src="#meshData">
    </group>
 </xml3d>
```

As soon as the new group node was inserted in the DOM, it will automatically be rendered in the 3D view. Transformations applied to group nodes can be changed as follows:

```
var transform = $(group.transform); // retrieve a transformation via
jQuery
transform.translation.set(new XML3DVec3(1,2,3)); // Translation and
Scale can be assigned as vectors by ".set()"
transform.rotation.set(new XML3DRotation(new XML3DVec3(0,1,0), //
Rotations can be assigned as axis angle by ".set()"
                      0.5*Math.PI));
```

Updating transformation attributes automatically renders the frame and displays the applied changes.

8.3.3.1 *Hardware Accelerated Parallel Processing with Xflow*

Xflow provides a possibility for effective parallel data processing on CPU or GPU device by utilising WebCL. This is especially useful if developers are using Xflow for processing big datasets. For smaller datasets the benefits are not so clearly visible.

Declaring a WebCL based Dataflow does not differ from ordinary Xflow declaration. If WebCL is available on users computer (by utilising Nokia's WebCL plugin on FireFox) WebCL-based data processing is automatically utilised by Xflow. Developers can also force the dataflow to utilise the WebCL platform by setting the optional "platform" attribute value to "cl".

Below is an example of how to declare a WebCL based dataflow and how to force the processing platform to be WebCL.

HTML:

```
    <dataflow id="blurImage" platform="cl">
        <compute>
            blur =  xflow.blurImage(image, 9);
        </compute>
    </dataflow>
```

However, in order to make the WebCL based data processing to work, a WebCL Xflow operator needs to be registered in a separate JavaScript script.

Below is an example of registering a WebCL Xflow operator. This operator applies a blur effect on the input "image" texture parameter and outputs the processed "result" texture.

Javascript:

```
Xflow.registerOperator("xflow.blurImage", {
    outputs: [
        {type: 'texture', name: 'result', sizeof: 'image'}
    ],
```

```
    params: [
        {type: 'texture', source: 'image'},
        {type: 'int', source: 'blurSize'}
    ],
    platform: Xflow.PLATFORM.CL,
    evaluate: [
        "const float m[9] = {0.05f, 0.09f, 0.12f, 0.15f, 0.16f, 0.15f,
0.12f, 0.09f, 0.05f};",
        "float3 sum = {0.0f, 0.0f, 0.0f};",
        "uchar3 resultSum;",
        "int currentCoord;",
        "for(int j = 0; j < 9; j++) {",
        "currentCoord = convert_int(image_i - (4-j)*blurSize);",
        "if(currentCoord >= 0 || currentCoord <= image_width *
image_height) {",
        "sum.x += convert_float_rte(image[currentCoord].x) * m[j];",
        "sum.y += convert_float_rte(image[currentCoord].y) * m[j];",
        "sum.z += convert_float_rte(image[currentCoord].z) * m[j];",
        "}",
        "}",
        "resultSum = convert_uchar3_rte(sum);",
        "result[image_i] = (uchar4)(resultSum.x, resultSum.y,
resultSum.z, 255);",
    ]});
```

The WebCL Xflow operator is designed in a way that allows a developer to focus purely on the core WebCL kernel programming logic. Developers can write their WebCL kernel code in the "evaluate" attribute of the operator, like shown in the example above. The code that can be written there is based in C language and the methods defined in the WebCL specification can be freely utilised. However, no kernel function headers or input/output parameters need to be defined as they are created automatically by the underlying Xflow architecture.

Xflow processes "outputs" and "params" of the Xflow operator and allows them to be directly used in the WebCL kernel code. As seen in the example above, the input parameter "image" can be directly used in the code. An iterator for the first input parameter is also automatically generated and it can be safely used in the code. For the "image" param the iterator variable is named as "image_i". Also, some helper variables such as "image_height" and "image_width" are generated and likewise, they can be used in the evaluate code. Only the texture type parameters have height and width helper variable because textures or images are a special case; they are two-dimensional data stored in one-dimensional buffer. All other input parameter types have a "length" helper variable e.g. "parameterName_length" that determines the length of the input buffer.

Additionally, all WebCL application code needed for executing the WebCL kernel code (such as passing WebCL kernel arguments to the WebCL program and defining proper WebCl workgroup sizes) is generated automatically. Thus, developers need no deep knowledge of the WebCL programming and basic programming skills are enough to produce kernel code for simple WebCL Xflow operators.

Below is an example of a very simple WebCL Xflow operator. This operator is used for grayscaling an input texture. Only three lines of kernel code is required.

```
Xflow.registerOperator("xflow.desaturateImage", {
       outputs: [
           {type: 'texture', name: 'result', sizeof: 'image'}
       ],
       params: [
           {type: 'texture', source: 'image'}
       ],
       platform: Xflow.PLATFORM.CL,
       evaluate: [
           "uchar4 color = image[image_i];",
           "uchar lum = (uchar)(0.30f * color.x + 0.59f * color.y +
0.11f * color.z);",
           "result[image_i] = (uchar4)(lum, lum, lum, 255);"
       ]
   });
```

9 3D-UI - WebTundra - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

9.1 Introduction

This documentation introduces the usage and development of WebTundra applications.

WebTundra is a WebGL & WebSockets based web client library for making multiuser applications by utilizing the realXtend Tundra SDK as the server. The server features 3d physics (with Bullet), Javascript scripting for server side logic and efficient networking with the kNet library. WebTundra provides the application developer with a scene model which is automatically synchronized over the network and vizualised in 3d in the browser client.

To make own applications you can use WebTundra as a Javascript library in your web application. The focus is on networked multiuser applications with (realtime) interaction / collaboration. The game of Pong is used as minimal but complete example of a multiuser application created on the platform. We use it to illustrate the use of the overall system: what is typically running on the server side and what kind of hooks the client library has for custom functionality. How you can create your own web application which uses WebTundra as a very high level library for networking.

Developer documentation for the underlying parts from the Synchronization GE: is documented in:

- Data model, the scene entity-system in WebTundra: Synchronization - Installation and Administration Guide#Data model
- The scene and networking library: Synchronization - User and Programmers Guide#JavaScript client library classes

WebTundra and Three.js in general allow the whole application to be developed with software code, even generating the 3d geometry etc. However 3D models, animations and full scenes are often created in modelling applications, such as open source Blender or Autodesk's Max. Therefore a key part of 3D-UI is the asset pipeline from modelling applications, or possible other sources, to the runtime engine. Exporting of 3d creations from Blender is documented in detail in the User Guide section of this manual.

9.1.1.1 *Background and Detail*

These guides relate to the 3D-UI-WebTundra implementation of the 3D-UI Generic Enabler which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more details about this Generic Enabler in the according Architecture and Open Specification documents.

9.2 User Guide

People use WebTundra applications simply by heading to the website of an application. The end user interface is totally up for the application developer to define as always with web applications. The base template we provide for developing the application is https://github.com/realXtend/WebTundra/blob/master/app.html .

To create applications one typically needs to program the functionality in Javascript as in all Web development. That is covered in the next section of this manual. But it is possible to use WebTundra also to just publish 3d models or scenes on the web without any programming. Basic functionality can be introduced by using existing components such as a free-flying camera. This section of the manual documents how to export 3d models, animations and materials from a modelling application to be used in WebTundra or plain Three.js. We use the open source 3d creation suite Blender in the examples, but there are exporters for the same formats also for applications (such as Autodesk's Max and Maya).

We document and support two families of formats: 1. Three.js's native JSON files for both individual meshes (including animations) and full scenes and 2. Khronos group's work-in-progress standard proposal glTF (aka. Collada JSON). Main focus is on the Three.js's JSON formats as they are straightforward to use, mature, feature complete and most used in the three.js community. The glTF specification is not finished yet and the support for it is recent but it is relevant for two reasons: it is more efficient, and a well defined standard. First parst of this section detail the use of Three.js formats and the last separate section documents glTF use.

9.2.1 Exporting 3D scenes from authoring applications

To export animations and material definitions correctly they typically need to be setup in a certain way in the authoring application. We provide an example file as a reference, a white-tailed deer from realXtend's nature demo scene: https://github.com/realXtend/chesapeakebay/blob/master/objects/White_Tailed_Deer/White_Tail_Deer_all.blend

A ready-made export and other demos of the graphical features are available in https://github.com/playsign/WebTundraGFX .

Figure 1. White tail deer with normal map+diffuse+specular

9.2.1.1 ***Export from blender to three.js with skeleton animations***

Example: https://github.com/playsign/WebTundraGFX/blob/master/DeerDemo/ExportedShaderMaterial.html

- Three.js requires mesh to be unparented to the armature. Else you will notice clear graphical glitches. Unparenting in the deer blend: https://github.com/playsign/WebTundraGFX/blob/master/doc/BlenderSkeletonAnimation.PNG

- In Three.js export settings skinning and bones should be selected, check previous screenshot for an example.
- Instead of regular THREE.Mesh, we need to use THREE.SkinnedMesh

```
deer = new THREE.SkinnedMesh(geometry,material);
```

- Enable skinning in all materials:

```
function enableSkinning(skinnedMesh) {
    var materials = skinnedMesh.material.materials;
    for (var i = 0, length = materials.length; i < length; i++) {
        var mat = materials[i];
        mat.skinning = true;
```

- Add animations to animation handler

```
THREE.AnimationHandler.add( geometry.animations[0] );
```

- Play animation

```
var animation = new THREE.Animation( deer, "Run-1-15" );
animation.play();
```

- Update animation handler with delta time

```
THREE.AnimationHandler.update( delta );
```

9.2.1.2 *Export from blender to three.js with morph target animations*

Examples:

http://www.stickmanventures.com/labs/demo/webgl-threejs-morph-target/

http://www.stickmanventures.com/blog/2011/09/07/simple-facial-rigging-utilizing-morph-targets-powered-by-three-js/

http://threejs.org/examples/webgl_morphtargets.html

- From timeline editor set start and end frames. Frames between them will get exported. If you don't know how many frames to select then open NLA editor to see animation tracks.
- Export with Blender exporter for Three.js. From export settings select (at least) vertices, faces, normals, UVs, Flip YZ, Embed meshes and Morph animation. If you want to export whole blender scene then also select All meshes, otherwise it will export selected mesh.

9.2.1.3 *Export from blender to three.js with normal map+diffuse+specular*

Example:
https://github.com/playsign/WebTundraGFX/blob/master/DeerDemo/ExportedShaderMaterial.html

- Preset a normal map texture to your material. Screenshot from the brain demo: https://github.com/playsign/WebTundraGFX/blob/master/doc/BlenderNormalmap.png

Mandatory settings are [v] Normal map and a source file.

- Preset a specular map texture. Select source file, uv mapping, diffuse color off, intesity to 1, rgb to intesity.

Example settings:
https://github.com/playsign/WebTundraGFX/blob/master/doc/BlenderSpecularmap.png

- Export with three.js exporter. Make sure Normals is checked in the export settings.

- In javascript code include:

```
   var jsonLoader = new THREE.JSONLoader();
   // addModelToScene function is called back after model has loaded
   jsonLoader.load( "models/Deer_with_specular_and_normal_mapping.js",
addModelToScene );
   {
        var material = new THREE.MeshFaceMaterial( materials );
        model = new THREE.Mesh( geometry, material );
        model.scale.set(10,10,10);
        scene.add( model );
   }
```

This way the material works automatically and no special Javascript code is required to configure it. If there are meshes on the server side scene their materials work, and the declarative xml3d authoring works also without material configuration code just by declaring:

```
 <mesh src="models/Deer_with_specular_and_normal_mapping.js">
```

9.2.1.4 *Using glTF via COLLADA*

glTF is an alternative format: when using it no Three.js exports are required. Instead, glTF is the proposed transfer format for the pre-existing COLLADA standard for which there already are exporters in all applications. Also Blender comes with COLLADA export included (using the same OpenCollada library as Maya and Max exporters do). The glTF loader for Three.js is bundled and integrated to WebTundra so that it works directly.

glTF is created from COLLADA source by running a separate converter: COLLADA2GLTF . That converts your .dae file to a .json file and accompanying .bin and .glsl files. The built-in glTF loading in WebTundra relies on the glTF file to have .gltf extension, instead of the default .json, to differentiate from the three.js or possible other json mesh files. So after export you must rename the .json to .gltf. We reported this back to Khronos as a possible modification to the standard specification: https://github.com/KhronosGroup/glTF/issues/260

After that a glTF mesh reference in a mesh component works:

```
<mesh id="duck" src="duck.gltf"></mesh>
```

A full working example of that minimal loading of the example Duck mesh with shaders is in examples/gltf/

For more information about glTF and downloads for the converter see http://gltf.gl/ (note: the spec is still living so binary releases may not be up-to-date with the development code)

9.2.2 How to use avatar in WebTundra

Example: https://github.com/realXtend/WebTundra/tree/dev/examples/Avatar

Here is an example how to create avatar in WebTundra:

```
var ent = scene.createEntity(0);
ent.createComponent(0, "Placeable");
var avatar = ent.createComponent(0, "Avatar");
```

```
var ref = avatar.appearanceRef;
ref.ref = ”avatar.json”;
avatar.appearanceRef = ref;
```

In EC_Avatar component there are two ways to load JSON description object

- Loading using avatarRef

```
var ref = avatar.appearanceRef;
ref.ref = ”avatar.json”;
avatar.appearanceRef = ref;
```

- Directly pushing JSON object to avatar

```
avatar.setupAppearance( jsonData );
```

Avatar description object can hold following variables:

- name: Entity name
- geometry: Reference to mesh asset (EC_Mesh).
- transform: Contains position, rotation (euler angles), scale (EC_Placeable).
- materials: List of materials used by the geometry. Note! not supported since EC_Mesh component material attribute isn't implemented yet.
- animations: If animation is defined, EC_Avatar will add EC_AnimationController component to an entity but animation name and src variables are ignored.
- Parts: List of child objects that can have similar variables as the root object, but they can't hold their own parts.

If you want to attach part to a parent bone you can use transform parentBone variable in transform element:

```
"transform" : {
    "pos": [0, 0, 0],
    "rot": [90, 0, 0],
    "scale": [1, 1, 1],
    "parentBone": "hand.L"
}
```

Here is an example of avatar description file:

```
{
    "name"      : "RobotAvatar",
    "geometry"  : "robot.json",

    "transform" :
```

```
    {
        "pos": [0, 0, 0],
        "rot": [0, 0, 0],
        "scale": [1, 1, 1]
    },

       "materials" :
    [
        "submesh1_materialref",
        "submesh2_materialref"
    ],

    "parts" :
    [
        {
            "name"      : "Sword1",
            "geometry"  : "Sword.json",

            "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [90, 0, 0],
                "scale": [1, 1, 1],
           "parentBone": "hand.L"
            },

            "materials" :
            [
                "submesh1_materialref",
                "submesh2_materialref"
            ]
        },
              {
            "name"      : "Sword2",
            "geometry"  : "Sword.json",

            "transform" :
            {
```

```
                "pos": [0, 0, 0],
                "rot": [90, 0, 0],
                "scale": [1, 1, 1],
           "parentBone": "hand.R"
            },

            "materials" :
            [
                "submesh1_materialref",
                "submesh2_materialref"
            ]
        },
              {
            "name"      : "Pants",
            "geometry"  : "robot_pants.json",

            "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [0, 0, 0],
                "scale": [1, 1, 1]
            }
        }
    ],

    "animations" :
    [
        {
            "name" : "Walk",
            "src"  : "stickman@walk.dae"
        },
        {
            "name" : "WaveHand",
            "src"  : "stickman@wavehand.dae"
        }
    ]
}
```

9.3 Programmer's Guide

This guide helps programmers to get started with developing WebTundra applications.

9.3.1 Minimal Example

Minimal example application available in the git repository:

- https://github.com/realXtend/WebTundra
  - html: https://github.com/realXtend/WebTundra/blob/master/app.html
  - javascript: https://github.com/realXtend/WebTundra/blob/master/src/app.js

Basic development setup help available in the Installation and Administration Guide.

In a nutshell you need the following:

- Tundra web server, global or local. Check this url for the Tundra server setup help: http://realxtend.org/documentation/
- txml scene running in the server e.g. http://playsign.tklapp.com:8000/WebTundra/examples/Physics2/scene.txml . It's recommended to run a simple scene first with small number of entities.
- WebTundra repository cloned to your computer https://github.com/realXtend/WebTundra
- local web server in your computer so you can run applications locally on a web browser e.g. http://localhost:8000/GitHub/WebTundra/scenes/physics2/index.html

You can then start to program by modifying the html and javascript files (and server code if needed to).

9.3.2 Pong Example

A minimal but complete example application is multiplayer Pong implemented using WebTundra. It is available from https://github.com/playsign/PongThreeJS/tree/ec . Here we provide a simplified version of the code with explanations.

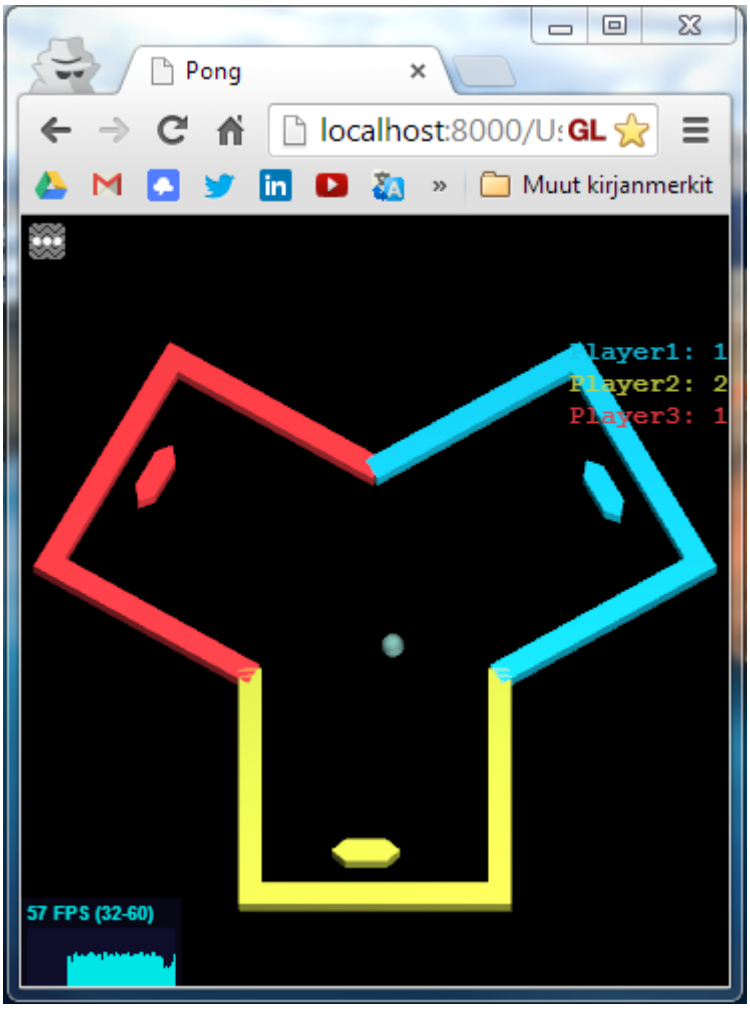


Figure 2. Three player online match

Server roles:

- Game control
- Player data control: amount of players, spectators and balls left (player lifes)
- Listen players connect and disconnect (11)
- Procedural scene generation
    - By loading single playerArea.txml x times (12)
- Physics (Bullet physics)
    - Everything is physics modeled: ball, rackets and walls
    - Normalize ball velocity (13)

Server-side code (gameController.js):

```
//(11)
if (server.IsRunning()) {
    server.UserConnected.connect(ServerHandleUserConnected); //(11a)
```

```
    server.UserDisconnected.connect(ServerHandleUserDisconnected);
//(11b)

    var users = server.AuthenticatedUsers();
    if (users.length > 0)
        console.LogInfo("[Pong Application] Application started.");

    for (var i = 0; i < users.length; i++) {
        ServerHandleUserConnected(users[i].id, users[i]); //(11c)
    }
}
...
function ServerHandleUserConnected(userID, userConnection) { //(11d)
    console.LogInfo("username: " + userConnection.Property("name"));
    console.LogInfo("player amount: " + playerAmount);
...
//(12)
    // Create player areas
    for (var i = 0; i < playerAmount; i++) {
        var areaParent = loadPart(partfile); //(12)
...
// Load the scene txml (playerArea.txml)
function loadPart(partfile) { //(12)
    var ents = scene.LoadSceneXML(pathForAsset(partfile), false, false,
2); //, changetype);

    // Set the racket ref in the parent entity
    var parentEntity = ents[0];

    var children = parentEntity.Children();

    // Save entity references for later use
    var areaComp = parentEntity.Component("PlayerArea");
    areaComp.racketRef = children[2].id;
    areaComp.borderLeftRef = children[1].id;

    return parentEntity;
}
...
```

```
//(13)
function update(dt) {
    var rigidbody = ball.rigidbody;
    var velvec = rigidbody.GetLinearVelocity();
    var curdir = velvec.Normalized();
    velvec = curdir.Mul(ballSpeed);
    rigidbody.SetLinearVelocity(velvec);
}
```

Client roles:

- Custom orthographic camera. Its position and orientation (1)
- Racket controls (2)
    - Client gets updated entity positions and orientations (ThreeView.js)
    - Racket control by modifying rigidbody velocity (14)
- Unique ID generation (15)
- Three.js scene control (ThreeView.js)
- User interface
    - Menus
    - Buttons
    - jQuery dialogs (10)
    - Ball amount
- Connect/disconnect
    - Triggered by buttons (16)

onlineButton.js:

```
...
    .button({
        label: "Connect"
    })
        .click(function() {
        // if not already connected
        app.sceneCtrl.showHelp();
        if (app.connected == false) {
            app.connect(app.host, app.port); //(16)
        }
    })
...
```

Initialize the app and define custom app properties :

```
var app;

function init() {
	app = new PongApp();
	app.host = "localhost"; // Address of the Tundra server
	app.port = 2345; // and port of the server

	function getRandomInt(min, max) {
		return Math.floor(Math.random() * (max - min + 1) + min);
	}

	// We have configured the app and so we are ready to start it
	app.start();

	// Custom app specific properties
	app.serverGameCtrl = undefined;
	app.racketSpeed = 80;
	app.reservedRacket = undefined;
	app.reservedPlayerArea = undefined;
	app.reservedBorderLeft = undefined;
	app.playerAreaWidth = 100;

	app.dataConnection.loginData = {
		"name": Date.now().toString() + getRandomInt(0,
2000000).toString() //(15)
	};
	 ...
```

Application constructor:

```
   function PongApp() {
	Tundra.Application.call(this); // Super class
    }
    PongApp.prototype = new Tundra.Application();
    PongApp.prototype.constructor = PongApp;
```

Define custom application logic init (mainly constructor calls):

```
    PongApp.prototype.logicInit = function() {
```

```
        // TOUCH
        this.touchController = new TouchInputController();
        // CAMERA
        var SCREEN_WIDTH = window.innerWidth;
        var SCREEN_HEIGHT = window.innerHeight;
        var NEAR = -20000;
        var FAR = 20000;
        // override camera
        this.camera = new THREE.OrthographicCamera(-SCREEN_WIDTH / 2,
SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2, -SCREEN_HEIGHT / 2, NEAR, FAR);
//(1)
        ...
```

(1): In the pong game we don't want to use default application perspective camera.

Define custom application update loop with entity manipulation:

```
    PongApp.prototype.logicUpdate = function(dt) {
        if (this.connected) {
            // RACKET CONTROL
            //(2)
            if (this.reservedRacket !== undefined &&
this.reservedPlayerArea.placeable !== undefined &&
(this.keyboard.pressed("left") || this.keyboard.pressed("right") ||
this.keyboard.pressed("a") || this.keyboard.pressed("d") ||
this.touchController.swiping /*&& delta.x !== 0)*/ )) {
                // Get racket's direction vector
                // Radian
                var rotation =
(this.reservedPlayerArea.placeable.transform.rot.y + 90) * (Math.PI /
180);
                var racketForward = new THREE.Vector3();
                // Radian to vector3
                racketForward.x = Math.cos(rotation * -1);
                racketForward.z = Math.sin(rotation * -1);
                racketForward.normalize();
                // Racket's speed
                if (this.touchController.swiping) {
                    // Touch / Mouse
                    racketForward.multiplyScalar(this.racketSpeed *
this.touchController.deltaPosition.x * this.touchController.swipeSpeed
* -1);
                    if (Math.abs(racketForward.length()) >
this.racketSpeed) {
```

```
                        racketForward.normalize();
                        racketForward.multiplyScalar(this.racketSpeed);
                    }
                } else {
                    // Keyboard
                    racketForward.multiplyScalar(this.racketSpeed);
                }
                // Read keyboard
                if (this.keyboard.pressed("left") ||
this.keyboard.pressed("a")) {
                    racketForward.multiplyScalar(-1);
                }
                // Set a new velocity for the entity
                this.reservedRacket.rigidBody.linearVelocity =
racketForward; //(14)
                // console.log(racketForward);
                // Inform the server about the change
                this.dataConnection.syncManager.sendChanges();  //(3)
            }
            // Players info
            if (this.serverGameCtrl) {

this.sceneCtrl.refreshPlayersInfo(this.serverGameCtrl.componentByType("
PlayerAreaList").areaList.length);
            }
        }
    };
```

(2): Here we need to make sure that this.reservedRacket and this.reservedPlayerArea entities exist.

(3): After we manipulated entities we can send the new entity data to the server via WebTundraModel's synchronization manager.

Define WebSocket connection callbacks:

```
PongApp.prototype.onConnected = function() {
       Tundra.Application.prototype.onConnected.call(this);


       // Set callback function
       this.dataConnection.scene.actionTriggered.add(this.onActionTrigge
red.bind(this)); //(8)
};
```

```
PongApp.prototype.onDisconnected = function() {
	Tundra.Application.prototype.onDisconnected.call(this);

	// DESTROY SCENE OBJECTS
	var removables = [];
	var i = 0;
	for (i = 0; i < this.scene.children.length; i++) {
		if (this.scene.children[i] instanceof THREE.Object3D) {
			removables.push(this.scene.children[i]);
		}
	}

	for (i = 0; i < removables.length; i++) {
		if (!(removables[i] instanceof THREE.PointLight ||
removables[i] instanceof THREE.DirectionalLight || removables[i]
instanceof THREE.PerspectiveCamera || removables[i] instanceof
THREE.OrthographicCamera)) {
			removables[i].parent.remove(removables[i]);
		}
	}

	// Reset entity references
	this.serverGameCtrl = undefined;
	this.reservedRacket = undefined;
	this.reservedPlayerArea = undefined;
	this.reservedBorderLeft = undefined;
	this.dataConnection.scene.entities = {};
};
```

Get entity references and custom component attributes from the server

```
PongApp.prototype.getEntities = function() {
	console.log("getEntities");

	this.reservedRacket = undefined;
	this.reservedPlayerArea = undefined;

	// Find a player area that matches with the player
```

```
        this.serverGameCtrl =
this.dataConnection.scene.entityByName("GameController"); //(4)
        var areaList =
this.serverGameCtrl.componentByType("PlayerAreaList").areaList;
        var scene = this.dataConnection.scene;
        for (var i = 0; i < areaList.length; i++) {
                var entityID = areaList[i];
                var entity = scene.entityById(entityID);
                if (!entity) {
                        throw "entity not found";
                }
                var areaComp = entity.componentByType("PlayerArea");
                if (areaComp.playerID ==
this.dataConnection.loginData.name) {
                        // Set player area entity references
                        var racketRef = areaComp.racketRef; //(5)
                        var borderLeftRef = areaComp.borderLeftRef;
                        this.reservedRacket = scene.entityById(racketRef);
//(6)
                        this.reservedBorderLeft =
scene.entityById(borderLeftRef);
                        this.reservedPlayerArea = entity;

                        break;
                }
        }

        if (this.reservedPlayerArea !== undefined) {
                this.setCameraPosition(areaList.length);
        }
};
```

(4): Here we get a reference to server's game controller entity (the entity with ID 2 in figure 3.). GameController knows all player areas (pong room halfs).

(5): Get racket entity reference from the player area's custom component (entity-component editor in figure 3)

(6): Use the entity id to get actual entity data.

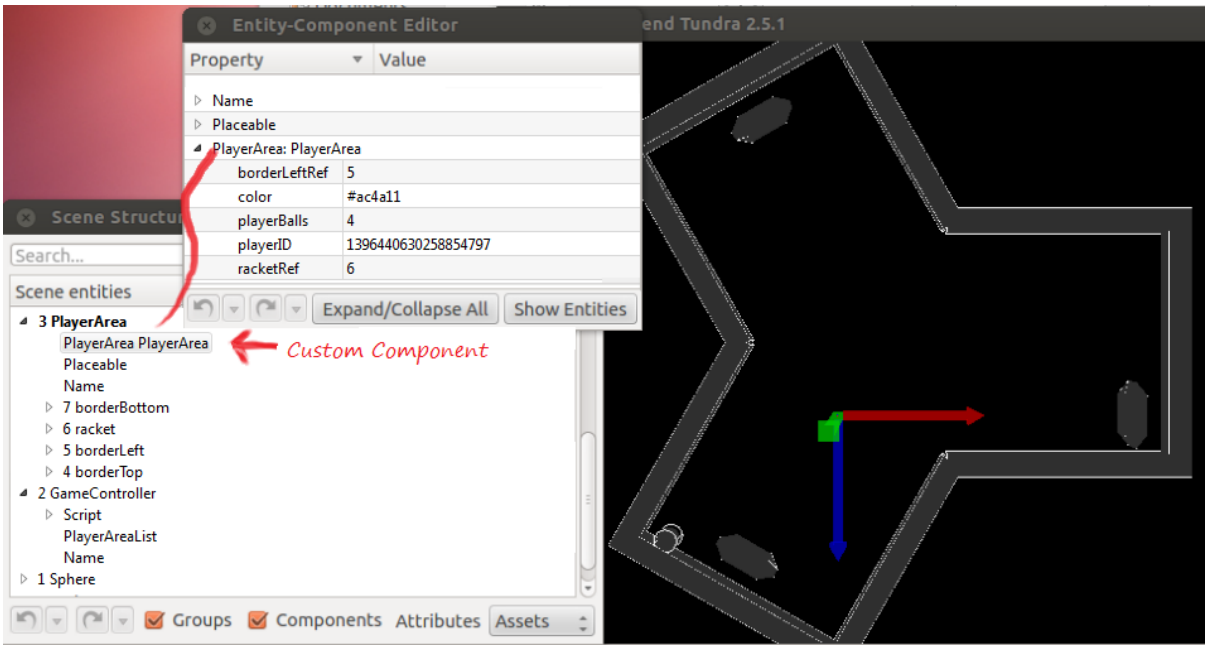


Figure 3. Pong server running on Tundra

Sending and receiving entity actions from a tundra server.

```
    Server-side:    (
https://github.com/playsign/PongThreeJS/blob/ec/server/gameController.j
s )
        // Notify clients about the player game over
        gameController.Exec(4, "gameover", "gameover", playerID,
playerAmount + 1); //(7)


    Client-side:
        // Set callback function

this.dataConnection.scene.actionTriggered.add(this.onActionTriggered.bi
nd(this)); //(8)
        ...
        // Action triggered callback
        PongApp.prototype.onActionTriggered = function(scope, param2,
param3, param4) {
              console.log("onActionTriggered");


              if (param2 === "sceneGenerated") {
                    // The scene is (re)generated
                    this.getEntities();
              }
```

```
                // Someone lost the game
                else if (param2 === "gameover") {
                        this.gameOver(param3[1], param3[2]);
                }
        };
        ...
        // Game over callback
        PongApp.prototype.gameOver = function(playerID, placement) {
//(9)
                var createDialog = function(dialogText) { //(10)
                        // jQuery dialog
                        var newDialog = 123321;
                        $("body").append("<div id=" + newDialog + "
title='Game Over'>" + dialogText + "</div>");
                        $("#" + newDialog).dialog({
                                width: 300,
                                height: "auto",
                        });
                };

                if (playerID == this.dataConnection.loginData.name) {
                        createDialog("You're out of balls. Placement: " +
placement);
                } else if (placement == 2) {
                        // We have a silver medalist and it's not you
                        createDialog("You won!");
                }
        };
```

(7): The server notifies clients about player game over (any player who loses the game and becomes a spectator). Clients need to know player id and player placement (playerID, playerAmount + 1)

(8): And here the client sets a callback for notifications from the server. gameOver function will get called via onActionTriggered (9)

(10): Simple jQuery dialog gets created when player's game ends (figure 4)

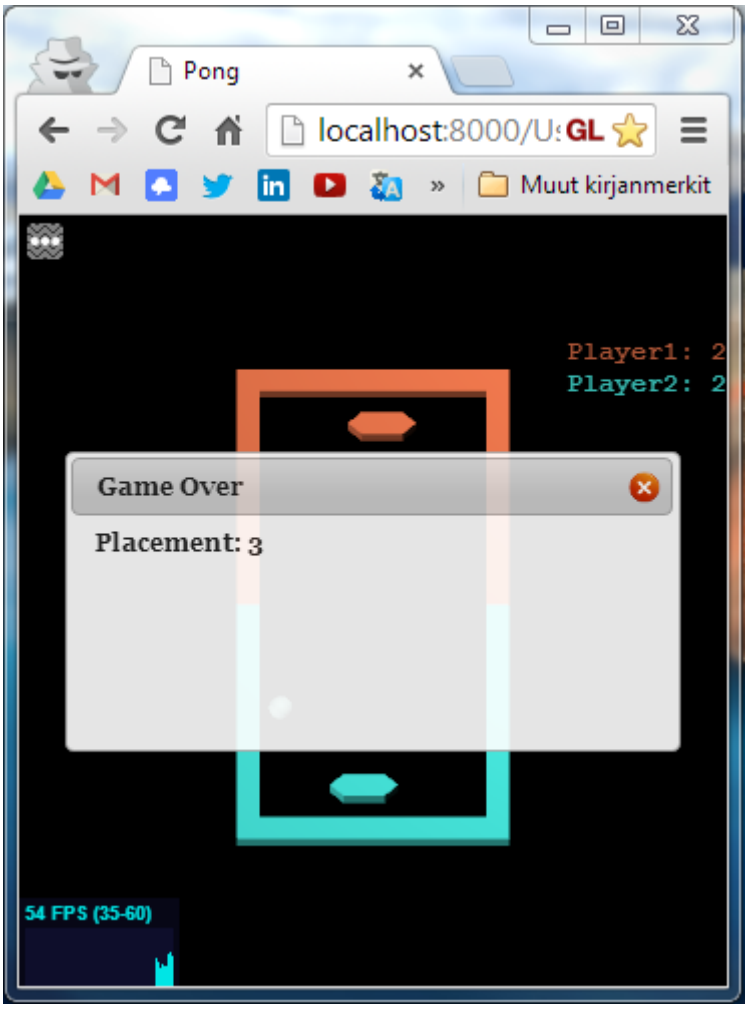


Figure 4. Game over jQuery dialog window

9.3.3 Physics example

We made a physics example app demonstrating the use of client-side physics via the Physijs library. It's available at the git repository https://github.com/playsign/WebTundraCar

In this example the user steers the car in a simple boxed scene, with car behaviour governed by a Physijs-based physics simulation.

9.3.3.1 *Running the physics example*

The source code for the top-level app is in the file example.js and the implementation of the car and Physijs integration is in car/car.js . The server-side code is in the server/ directory.

To try out the example, first follow the README to fetch the WebTundra library, and then make the WebTundraCar directory available over http, for example by running python -m SimpleHTTPServer in the WebTundraCar directory. Then start the Tundra server, like so: Tundra --server --file /where/i/put/my/WebTundraCar/server/scene.txml

Then, set the address of the Tundra server in the beginning of example.js and point your browser to where you configured the web server. To see a multi-user example, browse to the same address from another computer.

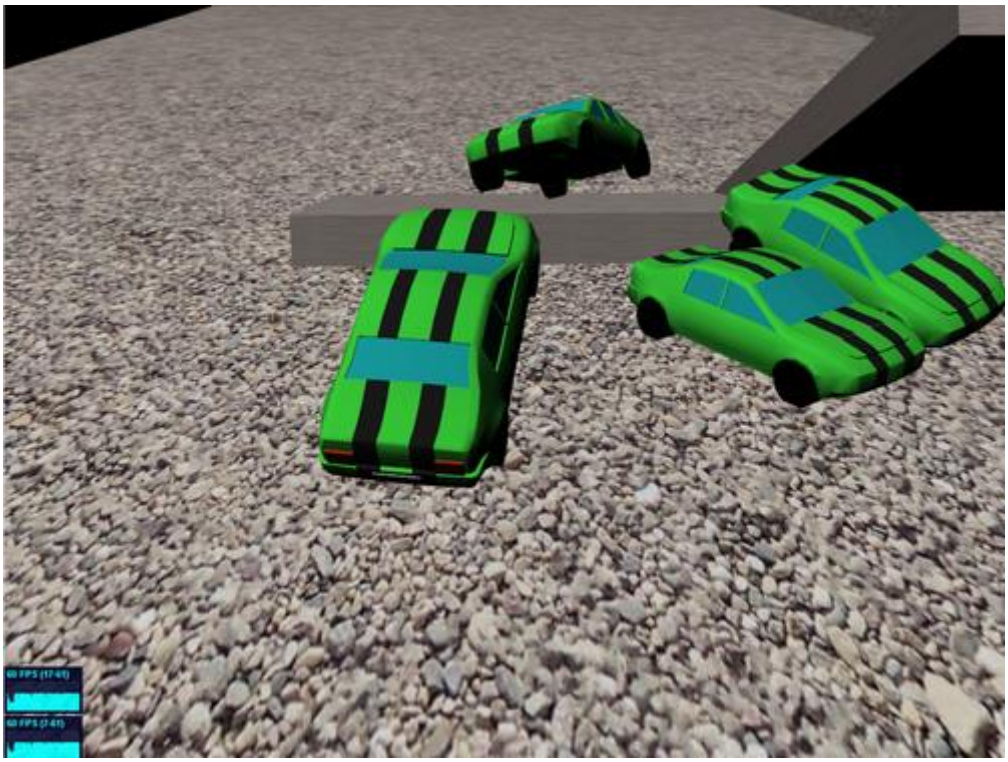


Figure 5. Physics example with cars

9.3.3.2 *How it works*

The relevant parts of the Physics example are the networking implementation and the Physijs integration.

Physijs integration works by overriding ThreeView to use the Physijs-provided Physijs.Scene class instead of THREE.Scene. Physijs.Scene is a THREE.Scene subclass but hooks up the physics simulation to the Three.js scene graph. After this inserting Physijs objects to the scene get physics simulation applied to them. For the car simulation, we use the Physijs-provided Physijs.Vehicle class. You can read more about Physijs at http://chandlerprall.github.io/Physijs/ and study the Vehicle physics related parameters in the car.js source file.

For the networking, the EC system is used. Each car is represented by an entity. Each car entity has a component representing its physics parameters. Parameters such as velocity, engine force, and braking force are synced over the network attriutes of entities, one entity per each user's car.

10 Synchronization - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

10.1 Introduction

This document describes the use of the Synchronization Generic Enabler, both the server and client parts.

The typical use of this GE is creating multi-user networked scenes, which may include 3D visualization (for example a virtual world or a multiplayer game). A client connects, using the WebSocket protocol, to a server where the multi-user scene is hosted, after which they will receive the scene content and any updates to it, for example objects' position updates under a physics simulation calculated by the server. The clients interact with the scene by either directly manipulating the scene content (this may not be feasible in all cases and can be prevented by security scripts running on the server) or by sending Entity Actions, which resemble remote procedure calls, that can be interpreted by scripts running on the server. For example, the movement controls of a client controlled character ("avatar") could be sent as Entity Actions (ie. move forward, stop moving forward, rotate 45 degrees right.)

Another use case is using the SceneAPI REST service to do infrequent scene queries and modifications. This has the advantage of not having to use the synchronization client library, and to not tie system resources to maintaining a real-time connection on either the server or the client.

Note that the Synchronization GE client code itself does not visualize anything, it only updates the internal scene data model according to data from the network. However, the same WebTundra codebase that houses the Synchronization client code also contains an implementation of the 3D-UI GE, which implements 3D visualization on top of the scene data model. Its use is described in 3D-UI - WebTundra - User and Programmers Guide.

10.1.1.1 *Background and Detail*

The Synchronization GE is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

10.2 User guide

The user guide section describes on a high level the data model used by this GE, and how scenes are loaded into the server.

10.2.1 Data model

The data that is transmitted between the server and the client is based on the Entity-Component-Attribute model as used by realXtend Tundra. It is described here https://github.com/realXtend/tundra/wiki/Scene-and-EC-Model

In particular, the following actions between the server and the client are synchronized bidirectionally through the WebSocket connection:

- Create an entity
- Remove an entity
- Create a component into an entity
- Remove a component from an entity

- Modify an attribute value in a component. This is the bulk of the network traffic. For example when an entity moves, the Transform attribute of its Placeable component is being updated constantly.
- Create an attribute into the DynamicComponent type of component.
- Remove an attribute from a DynamicComponent.
- Send an Entity Action.

When multiple clients are connected, such modifications are sent to all of them.

Note that the components that exist in the scene must have a corresponding JavaScript and C++ implementation that describes their static-structured attributes. That the set of attributes is known and fixed makes an efficient binary synchronization mechanism possible. On the other hand flexibility to freely describe data and new components on the fly is lost. For dynamic, application-specific data, the DynamicComponent should be used.

JavaScript implementations of often used components like Placeable are already provided by the client library code. Note that they don't by default implement any eg. rendering functionality, they only act as data containers.

The SceneAPI REST service offers an XML-based view into the scene, and provides the same operations as the real-time synchronization protocol, except for not providing Entity Actions, which are of a real-time nature:

- Query for all entities in the scene
- Query for an entity, component, or attribute
- Create an entity, with initial data or not
- Remove an entity
- Create a component into an entity, with initial data or not
- Remove a component from an entity
- Modify one or more attribute values in a component.

10.2.2 Using the Tundra server

The realXtend Tundra SDK is a complex piece of software and to provide a full description of how it can be used is beyond the scope of this document. For more documentation please see http://doc.meshmoon.com/ Note that MeshMoon is a proprietary hosting solution based on Tundra SDK but it currently provides the best, most up-to-date documentation of the Tundra SDK. Parts of the documentation which pertain only to the MeshMoon specific extensions and not the core Tundra SDK itself are marked so.

To see all Tundra's command line options use the command

```
Tundra --help
```

The following command line options are often used:

```
--file <scenename.txml> Specify the txml scene file to open on startup.
Example scenes exist in the
                        bin/scenes directory.
--server                Run as a server
--port <portnumber>     Specify which port the server listens on. This
is both for native clients
                        (UDP protocol) and WebSocket clients (TCP
protocol)
```

```
--headless                Run without graphics rendering
--httpport <portnumber> Enable Tundra HTTP server, which provides the
SceneAPI REST service. This
                          requires the HttpServerModule to be loaded
--config <configfile>   Specify the JSON configuration file(s) to use.
If not specified, the default
                          configuration file tundra.json is used.
```

10.2.3 Example

Without going into actual programming, the raw scene data synchronization functionality can be demonstrated in the following way using the Tundra server and a web browser's JavaScript debugging console. This assumes that both the Tundra server and the client JavaScript library source code have been successfully installed on the same machine as described in Synchronization - Installation and Administration Guide

Start up the Tundra server with rendering enabled so that you can see the scene. Here we choose the Physics2 example scene which consists of a large number of physics objects and a ball suspended above them. The server mode is enabled so that clients can connect. Additionally the tundra-addons.json configuration file is loaded, which allows the HttpServerModule to be loaded. The SceneAPI REST service is bound to port 2346.

```
Tundra --config tundra.json --config tundra-addons.json --file
scenes/Physics2/scene.txml --server --httpport 2346
```

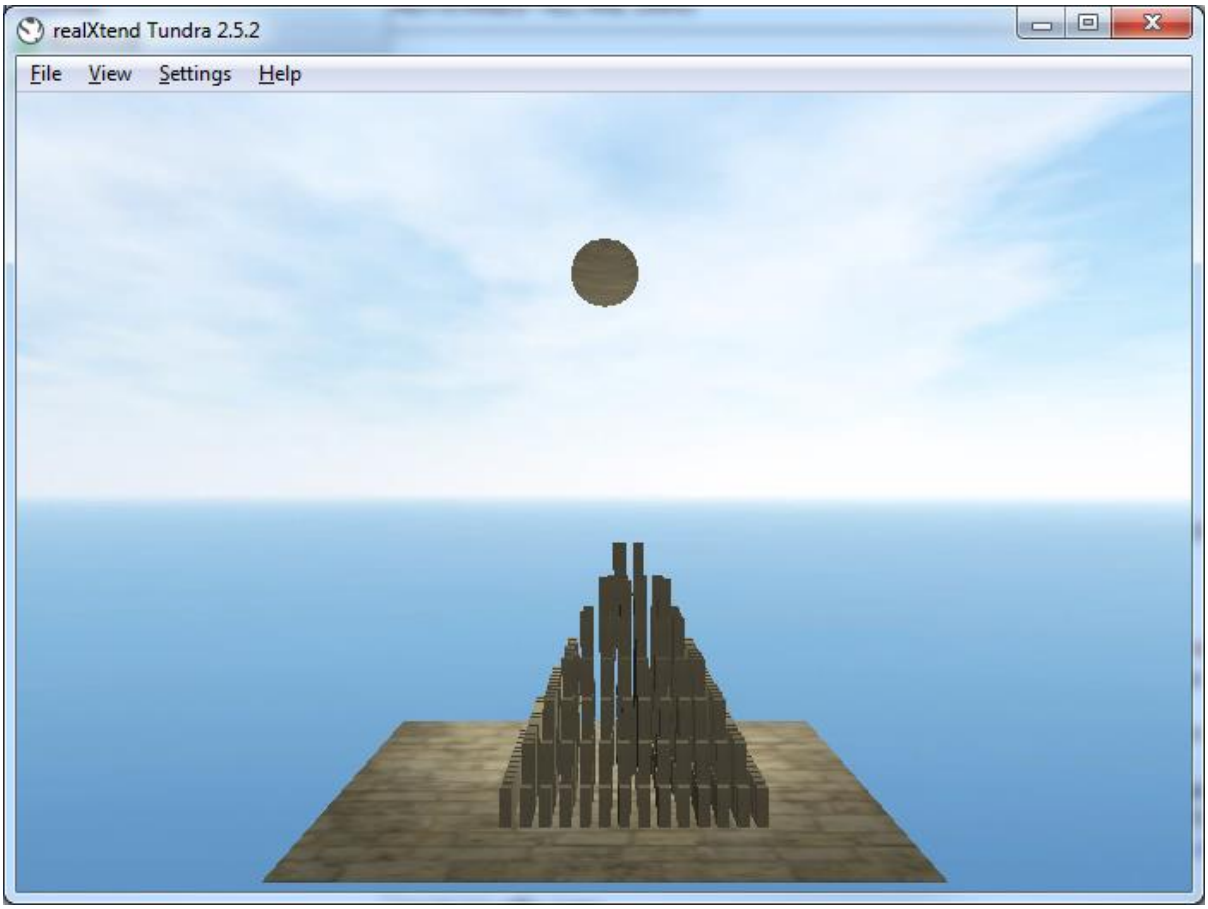
Note: if running the server from prebuilt binaries, the --config command line options can be omitted, as the tundra-addons.json configuration file is already being run by default.

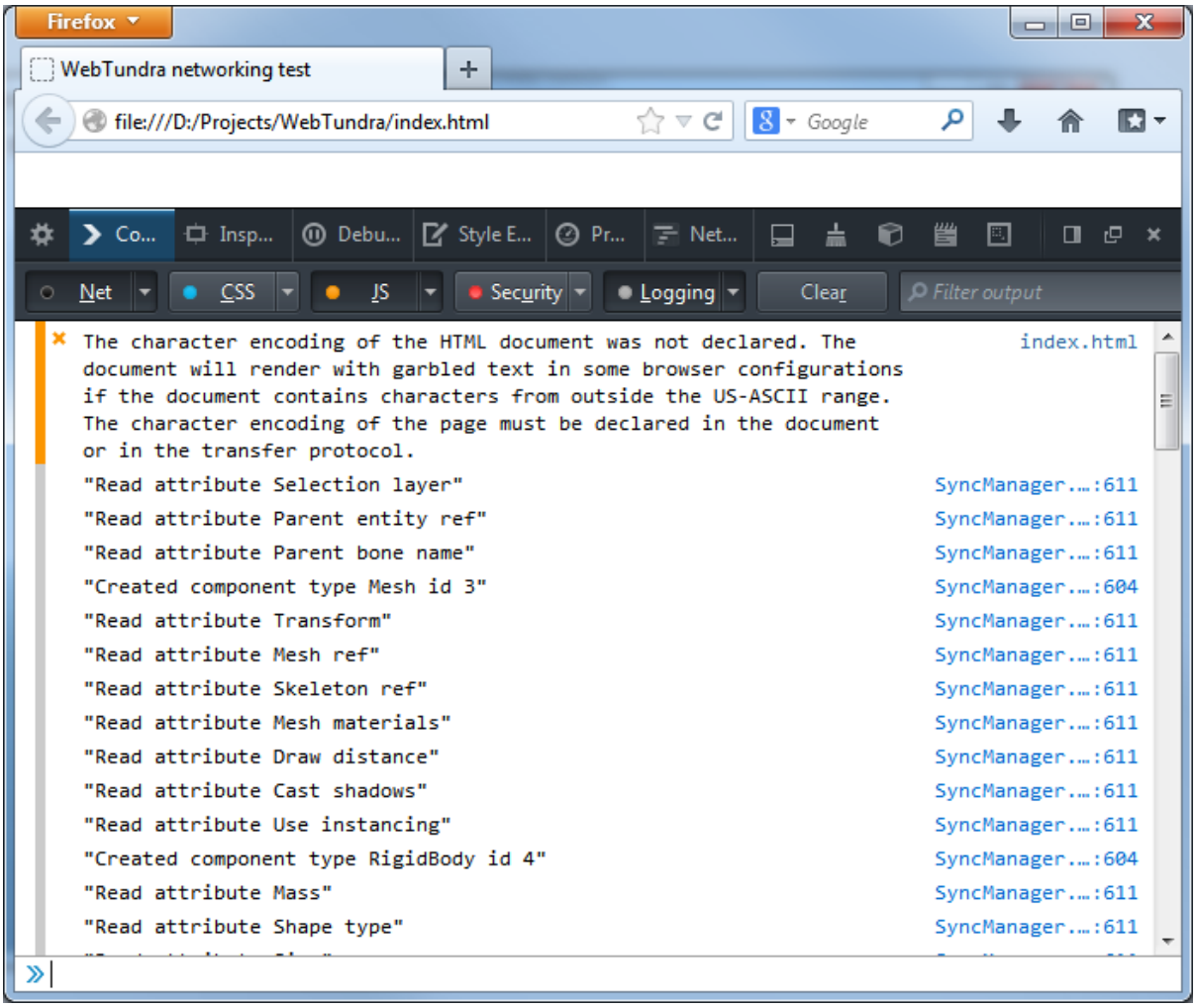
After this, open the file index.html with a web browser from the root directory of the client source code. Enable the browser's JavaScript console. The script application contained in index.html will automatically make a WebSocket connection to a localhost Tundra server with debug prints enabled, and you should see the initial scene data that has been received. The scene includes a moving light object, so you should continuously see this message being printed to the console, as the object moves:

```
Updated attribute Transform in component Placeable entity id 563
```

Now, click on the suspended ball above the objects in the server window. It will start to fall, colliding with the objects, and you will see even more console messages on the browser, as the other objects' position and velocity attributes will start replicating to the client as well.

The server (with rendering enabled) and browser windows should look like this when running properly. Note that it is intentional that the browser client displays debug information here instead of actually rendering the scene, as this GE is concerned of the data synchronization only.


realXtend Tundra 2.5.2
File View Settings Help



In a similar manner, the SceneAPI REST service can be demonstrated. The service requests use a URL path starting with /scene or /entities. The following GET request (can be executed with a browser) when the server is running, should display the whole scene's contents in XML format:

```
http://localhost:2346/entities
```

10.3 Programmers guide

The programmer's guide describes the JavaScript client library API, the SceneAPI REST service methods, and relevant parts of the server code's operation, as well as the binary-level protocol used.

10.3.1 JavaScript client library classes

All the client library classes are embedded within the namespace Tundra.

10.3.1.1 *WebSocketClient*

This class manages creating a WebSocket connection to the synchronization server and provides signals (which use the Signals.js library) for connection, disconnection and message received. It does not contain further logic.

To connect to a server, call

```
client = new Tundra.WebSocketClient();
```

```
client.connect(host, port, loginData);
```

where host is a hostname / IP address string, port is the TCP port to use (server is by default configured for 2345) and loginData is a JSON-formatted string containing application-specific login data, such as username or access credentials.

To close the current connection, call

```
client.disconnect();
```

To start crafting a binary network message, call

```
client.startNewMessage(msgId, maxBytes);
```

where msgId is the message ID (refer to the protocol documentation below) and maxBytes is the maximum size of data you expect to fill. The function returns a DataSerializer object which you can use to write binary data fields. Note that the message ID will automatically be written as the first data (a 16-bit integer) in the DataSerializer buffer.

To send the message contained in a DataSerializer object, call

```
client.endAndQueueMessage(dataSerializer)
```

After sending the dataSerializer object should not be reused, but should just be discarded. Call startNewMessage again to begin a new network message.

Messages from the server are signaled through the messageReceived signal. Its parameters are the message ID and a DataDeserializer object containing the binary message data. When the DataDeserializer is returned, the message ID has already been read from it. A skeletal example:

```
client.messageReceived.add(onMessageReceived);

function onMessageReceived(msgId, dataDeserializer) {

    // Check message type, for example using a switch-case. Then, if
it's a message ID

    // you are prepared to handle, start reading data from the
dataDeserializer

}
```

10.3.1.2 *SyncManager*

The SyncManager class implements reading scene synchronization messages from the server and modifying the client's scene accordingly, and also the reverse direction: when the client changes something in the scene that is to be replicated to the server, it will send the changes back.

When constructing, you need to give it a WebSocketClient object and a Scene object when constructing. It will then hook itself up to the necessary signals (network messages received, scene modifications happened)

For example this code would create an empty scene and connect to a localhost Tundra server running at the default port 2345, after which server-to-client scene synchronization would automatically commence.

```
var serverAddress = "localhost";

var serverPort = 2345;

var client = new Tundra.WebSocketClient();

var scene = new Scene();

var syncManager = new Tundra.SyncManager(client, scene);
```

```
client.connect(serverAddress, port);
```

The SyncManager will not automatically send changes to the server as they happen in the scene. Rather they will be collected in an internal data structure, allowing modifications to be batched. Call the sendChanges() function at steady intervals (for example 20 times per second) to actually send them:

```
syncManager.sendChanges();
```

10.3.1.3 *Scene*

The Scene class is a container for Entities, which can be queried for using their integer ID's.

The entity and component ID numbers are split into different ranges based on their purpose:

```
0x00000001 - 0x3fffffff Replicated entities / components. These are
synchronized between the server and the client

0x40000001 - 0x7fffffff Unacked entities / components. When a client
creates an entity or component that should appear

                        on the server, it allocates an "unacked" ID for
it. This will be transformed by the server

                        to an authoritatively allocated replicated ID.
A reply message is sent to the client so that

                        it knows also to change the ID of the entity or
component it created.

0x80000001 - 0xffffffff Local entities / components. These are created
on the client only (or the server only) and will

                        not be synchronized.
```

All scene operations (creation, modification and removal of entities, components and attributes) are accompanied by a "change type" or signaling mode. It is possible to make changes to the scene without sending them to the network, or even without signaling them internally at all. The default change type should be used in most cases, and is also chosen when the change

```
Tundra.AttributeChange.Default      Use the default signaling mode
which makes most sense, which is replicated for

                            replicated entities / components, or
localonly for local

Tundra.AttributeChange.Replicate    If the entity or component in
question is replicated, send the change as a network message

Tundra.AttributeChange.LocalOnly    Signal the change locally, but do
not send a network message

Tundra.AttributeChange.Disconnected Do not signal locally and do not
send a network message
```

To query an entity from the scene by its ID call

```
scene.entityById(id)
```

To create an entity to the scene, call

```
scene.createEntity(id, changeType)
```

To let the scene assign an ID, leave the ID zero. This should be used in most cases. If the change type is AttributeChange.Replicate, a replicated entity is created (like explained above, on

the client this involves the client choosing an unacked ID, and the server sending back the authoritatively allocated replicated ID) If the change type is AttributeChange.LocalOnly, a local ID will be assigned.

To return all scene entities that are unparented as an array, which are said to be at the scene root level (more on child entities below in the Entity section), call

```
scene.rootLevelEntities()
```

10.3.1.4 *Entity*

The Entity class is a container of components and child entities. Components also have an ID, which are unique only inside the same entity. They are also identified by their type.

To create a component to an entity, call

```
entity.createComponent(id, typeId, name, changeType);
```

Leave the ID zero to let the entity assign, should be used in most cases. TypeID is either the numeric typeID that is known for each component type for optimization in the network protocol, alternatively the string name (such as "Placeable" or "Mesh") can be used.

Components can optionally have names. Names are used to distinguish if the entity has multiple components of the same type.

The changetype (either AttributeChange.Replicate or AttributeChange.LocalOnly) will decide if the component will be network-synchronized or local (unsynchronized).

To delete a component by its ID, call

```
entity.removeComponent(id, changeType);
```

To look up a component, call

```
entity.componentById(id);
entity.componentByType(typeId);
```

where the typeID is either a number or a string type name.

There also exists a convenience property mechanism for accessing components. For example, if there are Placeable, Mesh and RigidBody components in an entity, they can be accessed with

```
entity.placeable
entity.mesh
entity.rigidBody
```

ie. the component typename's first letter is turned into lowercase.

Entity Actions are also sent through the Entity class. These consist of an action name as a string, a string array of parameters, and an execution type that tells whether to execute the action locally, on the server, on other connected clients (peers), or a combination of these.

```
entity.triggerAction(name, params, execType);
```

The execution type constants for Entity Actions are the following:

```
Tundra.cExecTypeLocal
Tundra.cExecTypeServer
Tundra.cExecTypePeers
```

10.3.1.5 *Component*

Components hold attributes. Subclasses of most used Tundra components exist in the JavaScript client library, but the amount of components implemented currently is not exhaustive.

For the subclasses, look for the code files beginning with the EC_ prefix, for example EC_Placeable.js implements the Placeable component.

A component's attributes can be accessed from the component's "attributes" array, where they are allocated consecutive indices starting from zero.

The attributes can also be accessed using convenience properties. For example the Transform attribute of a Placeable component is accessible as:

```
component.transform
```

The DynamicComponent is a special case component to which attributes can be created and removed at will. It contains no attributes by default. To create an attribute, call

```
dynamicComponent.createAttribute(index, typeId, name, value,
changeType);
```

The possible type IDs or type names are described below in the Attribute section. The value can be left null to not set an initial value.

To remove an attribute from a DynamicComponent, call

```
dynamicComponent.removeAttribute(index, changeType);
```

10.3.1.6 *Attribute*

Attributes hold the actual data contained by Components.

They are subclassed according to the attribute type, see the code file https://github.com/realXtend/WebTundra/blob/master/src/scene/Attribute.js for all the implementations.

The attribute types are: string, int, real, Color, float2, float3, float4, bool, uint, Quat, AssetReference, AssetReferenceList, EntityReference, QVariant, QVariantList, Transform, QPoint.

The structure of the attribute's value depends on the type, but it can always be accessed as the "value" property of the attribute.

If you modify the value property, the change will be propagated using AttributeChange.Default change type. To have control over the change type, call

```
attribute.set(newValue, changeType);
```

instead.

10.3.1.7 *Child entities*

From release 3.3 onward the scene model supports hierarchic composition of entities (parent-child relationships.) A parent entity "owns" its child entities so, that if the parent is removed from the scene, the child entities will also be removed.

To create a child entity, call

```
entity.createChild(id, changeType);
```

The created child entity is returned. Use ID 0 to let the scene assign the next free id. Use Tundra.AttributeChange.Default (same as omitting the changeType parameter) or Tundra.AttributeChange.Replicate as the changeType parameter to create a replicated child entity, and Tundra.AttributeChange.LocalOnly to create a local entity. You should not attempt to

create replicated child entities to a local parent entity; these will be skipped by the network synchronization.

An entity's immediate children are available as the JavaScript array

```
entity.children
```

To remove a child entity (and also remove it from the scene), call

```
entity.removeChild(childEntity)
```

To remove a child entity from the parent without removing it from the scene, but instead returning it to unparented state (root level), call

```
entity.detachChild(childEntity)
```

To reparent an entity to another parent entity, or to the scene root level (null parent), call

```
entity.setParent(newParentEntity)
```

The entity's current parent entity is available as the entity.parent property.

10.3.1.8 *Custom components*

From release 3.3 onward the scene model supports the concept of registering custom, static-structured component types, which do not need to have a C++ counterpart on the server, but are serialized on the network just as efficiently as those that do. The process of registration involves creating a "blueprint" component to which the desired attributes are created. This is more efficient than using the DynamicComponent, because the attributes and their types need to be specified only once, instead of specifying them for each instance. Each custom component type needs to have an unique type name in the scene. The information of custom component types is bidirectionally propagated between the server and the client.

An example of creating a custom component type with 3 attributes, creating a new entity with that component, then using the SyncManager to propagate both the custom type information, and the new entity to the server. Note that for attributes, both the id (variable name) and name (human-readable name shown in the Tundra scene structure window) can be specified.

```
var blueprint = new Tundra.Component();
blueprint.addAttribute("real", "engineSize", "Engine Size");
blueprint.addAttribute("real", "topSpeed", "Top Speed km/h");
blueprint.addAttribute("bool", "automatic", "Automatic Transmission");
Tundra.registerCustomComponent("Car", blueprint);
var ent = scene.createEntity(0);
ent.createComponent(0, "Car");
ent.car.engineSize = 1.6;
ent.car.topSpeed = 195;
syncManager.sendChanges();
```

10.3.2 SceneAPI REST service

When the Tundra server has been started with the SceneAPI REST service enabled (HttpServerModule is loaded and the --httpport command line parameter has been given to set up the port it should be bound to) the following requests are available. Note that /entities is interchangeable with /scene.

10.3.2.1 *Queries*

Query all entities in the scene. They are returned as XML data with a root 'scene' element similar to the .txml scene files

```
GET /entities
```

Query a specific entity from the scene by ID number. It is returned as XML data with a root 'entity' element.

```
GET /entities/id
```

Query a specific entity from the scene by its name. It is returned as XML data with a root 'entity' element.

```
GET /entities?name=EntityName
```

Query a specific component type (for example Mesh or Placeable) from an entity specified by ID. The component is returned as XML data with a root 'component' element.

```
GET /entities/id/componentTypeName
```

Query a specific attribute from a component from an entity specified by ID. The attribute value is returned as plaintext. For example /entities/1/placeable/transform

```
GET /entities/id/componentTypeName/attributeName
```

10.3.2.2 *Scene manipulation*

Replace an entity's data. Existing components will be removed. The data should be contained in the request body in XML format with a root 'entity' element. The updated entity XML data is sent back.

```
PUT /entities/id
```

Replace a component's attribute data. The attribute data should be contained in the request body in XML format with a root 'component' element. The updated component XML data is sent back.

```
PUT /entities/id/componentTypeName
```

Replace attribute value(s) in a component using query syntax. The updated component XML data is sent back.

```
PUT
/entities/id/componentTypeName?attributeName1=newValue1&attributeName2=
newValue2
```

Create a new entity while letting the server assign an ID. The entity's component data can optionally be contained in the request body in XML format with a root 'entity' element. The new entity's XML data (showing the proper server-assigned ID) is sent back.

```
POST /entities
```

Create a new entity with specific ID. If an entity with that ID already exists, the server will assign a new ID. The entity's component data can optionally be contained in the request body in XML format with a root 'entity' element. The new entity's XML data (showing the proper server-assigned ID) is sent back.

```
POST /entities/id
```

Create a new component to an entity. The component's data can optionally be contained in the request body in XML format with a root 'component' element. The new component's XML data is sent back.

```
POST /entities/id/componentTypeName
```

Remove an entity from the scene by ID.

```
DELETE /entities/id
```

Remove an entity from the scene by name.

```
DELETE /entities?name=EntityName
```

Remove a component from an entity.

```
DELETE /entities/id/componentTypeName
```

10.3.3 Server plugin

The server functionality of the Synchronization GE is implemented as two Tundra modules.

10.3.3.1 *WebSocketServerModule*

WebSocketServerModule implements the real-time synchronization protocol. Its code can be viewed here in Tundra's git repository: https://github.com/realXtend/tundra/tree/tundra2/src/Application/WebSocketServerModule

It uses the websocketpp library for implementing WebSocket communications (https://github.com/zaphoyd/websocketpp)

The WebSocketServerModule registers the WebSocket client connections to the Tundra main server class, so that other server-side modules and scripts can treat native (C++ client) and Web client connections as equivalent. The client list including both native and Web client connections can be received with one function call:

```
framework->GetModule<TundraLogic::TundraLogicModule>()->GetServer()->AuthenticatedUsers();
```

In server-side JavaScript code, this would be respectively:

```
server.AuthenticatedUsers();
```

10.3.3.2 *HttpServerModule*

HttpServerModule implements the SceneAPI REST service. Its code can be viewed as a part of the TundraAddons git repository: https://github.com/realXtend/TundraAddons/tree/master/HttpServerModule

This module also uses websocketpp library for the HTTP communications, though it is a modified version.

The HttpServerModule handles by itself requests that begin with URL path /scene or /entities. It does not handle other requests; instead these are emitted as a Qt signal by the HttpServer object; other C++ code can connect to this signal. To acquire the HttpServer object, use the following line of code:

```
framework->GetModule<HttpServerModule>()->GetServer();
```

See the file https://github.com/realXtend/TundraAddons/tree/master/HttpServerModule/HttpServer.h for the signal definition.

10.3.4 Synchronization binary protocol

For the description of the byte-level protocol see https://github.com/realXtend/tundra/wiki/Tundra-protocol.

- Each message is sent as one binary WebSocket frame, with the message ID encoded as an unsigned little-endian 16-bit value in the beginning.

- Login data (message ID 100) is JSON instead of XML.
- Before the server starts sending scene messages, the client must "authenticate" itself by sending the login message. In a default Tundra server configuration (no scene password, no security scripts) the actual data content sent in the login message does not matter.

11 Cloud Rendering - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

11.1 Introduction

This document describes the user and programmers guide of the reference implementation provided by the Cloud Rendering GE.

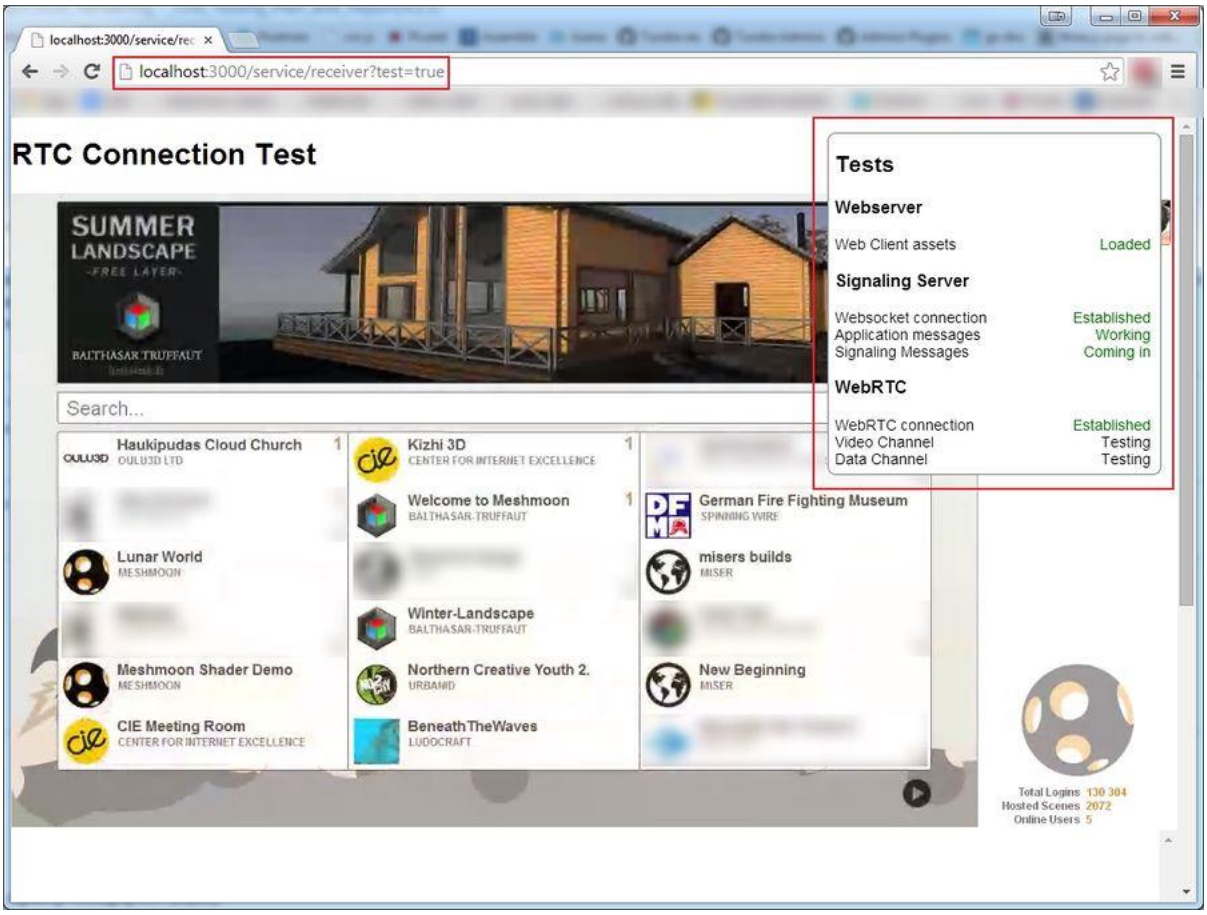
The Cloud Rendering GEi is split into three distinct components. Each of them will be covered, when appropriate, in the subsections of this page.

- **WebService** is a web service that does signaling between the renderer(s) and the web client(s). Facilitating the all important function of connecting two peers that want to communicate with WebRTC. Additionally application level messaging can be sent through it.
- **Renderer** is the application that delivers 3D rendering results to web clients via WebRTC video. In our case this is a realXtend Tundra based 3D virtual world client.
- **WebClient** is the application running in the end users web browsers that wants to receive 3D rendering from a renderer.

11.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Cloud Rendering GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

11.2 User guide



11.2.1 Renderer

The renderer is a plugin for the realXtend Tundra virtual world SDK. The software release of the Cloud Rendering GE contains a full Tundra distribution with the renderer plugin included. The following command line options to the Tundra executable are available if the plugins is loaded into Tundra.

11.2.1.1.1 *Command line parameters*

Loading the plugin to Tundra can be done by running `Tundra --plugin CloudRenderingPlugin`

The plugin can be configured using the following command line parameters

- `--cloudRenderer <web-service-host>` will instruct the plugin to act as a renderer and register itself to a **WebService** in the specified host.
- `--cloudRenderingSendWebCamera` will instruct the plugin to send web camera video instead of the 3D rendering to the clients.
- `--cloudRenderingShowStreamPreview` will make the renderer popup a rendering preview widget per client connection. This way you can see what actually gets sent to the web clients.

11.2.1.1.2 *Selecting a Tundra rendering plugin*

On Windows the default Tundra renderer is DirectX, however for the kind of render target blitting to a WebRTC stream is considerably faster with OpenGL. You can select the OpenGL renderer with `--opengl`

11.2.1.1.3 *Full execution example*

Example of combining the parameters to run a renderer against a localhost development web service

```
TundraConsole.exe --config tundra-client.json --opengl --plugin CloudRenderingPlugin --cloudRenderer localhost:3000
```

If you want to run with a Meshmoon Rocket configuration use the following. This gives you the full Meshmoon client that integrates into the Meshmoon hosting platform. If you are using the realXtend Tundra distribution you will additionally have to host your own virtual world server where you can login with the renderer. Meshmoon platform will offer a easier experience to find virtual worlds and to login into them.

`Rocket --opengl --plugin CloudRenderingPlugin --cloudRenderer localhost:3000`.

If the Cloud Rendering plugin was successfully loaded you will see the following line in the startup shell logging

```
 Loading plugin CloudRenderingPlugin
```

If it cannot connect to the web service

```
 Error: [WebRTC::Renderer]: WebSocket connection failed to Cloud
Rendering Service at ws://localhost:3000
```

11.2.1.1.4 *Debugging*

Add `--logLevel debug` to the application parameters to get verbose logging about the messaging between the web service and the web client(s). If you are connecting to virtual worlds with the client, you can add `--logLevelNetwork info` to reduce network related logging, which is not important in our context.

11.2.1.1.5 *Web Service notes*

The renderer will register itself to the web service. It is therefor mandatory that the web service is up and running when you start any renderers. In the above example they are ran locally on the same machines `localhost:3000` but the service can be ran anywhere and for any real world use desired due to the rather heavy CPU/GPU resource needs of the renderer.

11.3 Programmers guide

11.3.1 WebService

The web service is a ready server application that implements the full Cloud Rendering GE specification. There are little real world use cases where modifying the web service code would bring something new to your application. You are free to inspect the code to see how the protocol and specification is implemented and of course hack around the codebase. You can interact with the web service with the **WebClient** JavaScript library.

11.3.2 Renderer

Much of the above is also true for the renderer. The renderer is a specialized implementation of a Cloud Rendering renderer. Its function is to provide realXtend Tundra based 3D virtual world client to web clients. The code is open source and should be looked into with more detail if you are interested in implementing a Cloud Rendering renderer for your application. The application the renderer is providing is **NOT** limited to 3D rendering. You can stream WebRTC video and let the user interact with input events with any type of application.

If you are interested in the implementation details you can read the source code and inspect the doxygen documentation of the C++ code here.

11.3.3 WebClient

Installation is documented here. The web client example application and JavaScript library is built with the web service instructions.

Once you have built the web service, you can find the JavaScript library from `<web_service_source_code>/webservice/public/js/builds/RTCReceiver.js`. You must include this library on the page (web application) you are going to build with it.

11.3.3.1 *RTCReceiver.js*

The library is self configured and operates by default on the DOM of the page it is being instantiated on. It will by default try to connect to the same host as where the page is served from. This makes it work automatically for the purposes of the web service. However you can customize the behavior in the following ways.

11.3.3.1.1 *Creating a client*

```
// To instantiate a default client
var client = new CloudRenderingClient();
```

Connecting without options will use the following defaults

```
{
  host : 'ws://' + window.location.host,
  iceServers : [
    { 'url': 'stun:stun.l.google.com:19302' },
    { 'url': 'turn:130.206.83.161:3478' }
  ]
}
```

You can customize the web sevice host and used ICE servers with

```
var client = new CloudRenderingClient({
    host: "ws://my.service.com",
```

```
    username : "John Doe",
    iceServers : [
        { url : "my.stun.com" },
        { url : "my.turn.com" }
    ]
});
```

This client will establish a WebSocket connection with the web service at `host`.

11.3.3.1.2 *Sending and receiving messages*

You can override the default message handler function implementations. Handling these messages requires an understanding about the Cloud Rendering protocol that is documented on its own wiki page. What you decide to do with the message content also depends on your application that is using Cloud Rendering.

```
client.roomMessageHandler = function(message) {};
client.applicationMessageHandler = function(message) {};
client.signalingMessageHandler = function(message) {};
```

You can look into the `message` object structure from `<web_service_source_code>/signalingserver/lib/CRMessage.js`. This object can also be sent to the network by using.

```
// Channel is the protocols channel identifier. Type is the message
type inside the channel and payload is the message contents.
client.sendMessage(new CloudRenderingMessage(channel, type, payload));
```

11.3.3.1.3 *WebRTC interaction*

The WebRTC layer is handled by `<web_service_source_code>/jsapps/lib/PeerConnection.js`. All Cloud Rendering messages related to peer-to-peer WebRTC connection setup and execution is forwarded to this class. This part cannot be directly swapped from the provided JavaScript library, but can be done by modifying the web service source code and building it again. It is also a good place to check out if you are new to WebRTC JavaScript functionality.

12 Display As A Service - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

12.1 Introduction

This document describes the development of DaaS applications and what they offer to end-users. It describes how developers can use the Display As A Service API to add Virtual Displays to their applications, change the displayed content or manage the appearance for Virtual Displays on physical devices.

12.1.1.1 *Background and Details*

This guide describes the Display As A Service Generic Enabler of the Advanced Middleware and Web User Interfaces chapter. Please find detailed information in the respective Architecture and Open Specification documents.

12.2 User guide

End users of DaaS typically see consecutive live pixel streams in full synchronization across display walls of (potentially heterogeneous) independent devices only connected to a network, for example like this:

As DaaS is a framework and base technology for developing arbitrary applications, there are no "typical" UIs, pixels or projection arrangements you always see with DaaS. This is entirely up to the developer, which source applications to integrate in a DaaS session (i.e., what pixels will be sent around and ultimately shown somewhere), which sink applications (i.e., how incoming

streams are composited on which background on each connected display), and which controller applications (i.e., which pixels are mapped to which display, and what is the UI to enable this).

12.3 Programmers guide

In the following, we describe what is needed for DaaS application development, and what typical applications using the APIs of the DaaS SDK look like.

12.3.1 Programming DaaS Applications

In the following, ways to program with DaaS are described using code examples. The examples are ordered with respect to the DaaS entity (VFB, VD, Controller) they want to implement.

12.3.1.1 *VFB code example*

The following is a minimal C++application that uses the SDK at the Virtual Framebuffer (VFB) end of a DaaS session, i.e., for applications that produce pixels and provide them to the network. This example application just produces red frames with a test pattern and writes the pixels to a VFB. Everything else is handled by other components of the DaaS system (VDs to display them and Controllers to map them, respectively).

```
...

int main(int argc, char** argv)
{
        try
        {
                // Setup.

                const unsigned int w = 1920;
                const unsigned int h = 1080;
                const Format format(Format::UNSIGNED_BYTE,
Format::RGB);

                auto_ptr<VFB> vfb(VFB::create(w, h, format));
                cout << "Created VFB." << endl;

                // Main loop.

                const unsigned int frames = 1000;
                cout << "Rendering " << frames << " red frames: ";
                cout.flush();

                int m = 8;
                int x = 0;
                for (unsigned int count = 1; count <= frames; ++count)
```

```
                {
                        unsigned char* buffer = vfb->getBuffer();

                        // Write single color (red) to entire buffer.
                        for (unsigned int i = 0; i < w * h; i++)
                        {
                                buffer[i*3    ] = 255 * ( (i+x)%m != 0
);  // RED
                                buffer[i*3 + 1] = 255 * ( (i+x)%m == 0
);  // GREEN
                                buffer[i*3 + 2] =   0 * ( (i+x)%m == 0
);  // BLUE
                        }
                        if(x < 8) x++;
                        else x = 0;
                        vfb->signalEOF();

                        cout << count << " " << flush;
                }
                cout << endl;

                return EXIT_SUCCESS;
        }
        catch (NetVFB::Exception& e)
        {
                std::cout << "Caught NetVFB exception: " << e <<
std::endl;
        }
        catch (...)
        {
                std::cout << "Caught unknown exception." << std::endl;
        }

        return EXIT_FAILURE;
}
```

12.3.1.2 *VD code example*

The VD-sided API of DaaS (i.e., the interface for applications wanting to receive pixels in a DaaS session) is currently under development, so as of yet, there is no meaningful C++ example that does anything beyond creating a VD object and starting it.

12.3.1.3 *Controller code example (C++)*

The following is an example for C++ code using the DaaS Controller API. It creates a Controller instance and assigns several callbacks to it, depending on what changed in the system. In this case, whenever a different controller changes the properties of VFBs, VDs, or Projections, a callback is executed that outputs a console message.

```
...

class CallbackHandler
{

public:

        CallbackHandler(NetVFB::Controller* controller)
        : m_controller (controller)
        {
        }

        ~CallbackHandler()
        {
        }

        void displayChangeCallback()
        {
                std::cout << "Displays changed" << std::endl;
        }

        void projectionChangeCallback()
        {
                std::cout << "Projections changed" << std::endl;
        }

        void vfbChangeCallback()
        {
                std::cout << "VFBs changed" << std::endl;
        }

        }

private:
```

```
        NetVFB::Controller* m_controller;
};



int main(int argc, char** argv)
{
        try
        {
                 // Setup.
                std::auto_ptr<NetVFB::Controller>
controller(NetVFB::Controller::create("Controller"));


                //Register callbacks
                CallbackHandler ch(controller.get());
                controller->registerDisplayChangeCallback   (
std::bind( &CallbackHandler::displayChangeCallback,    std::ref( ch ) )
);
                controller->registerProjectionChangeCallback(
std::bind( &CallbackHandler::projectionChangeCallback, std::ref( ch ) )
);
                controller->registerVFBChangeCallback       (
std::bind( &CallbackHandler::vfbChangeCallback,        std::ref( ch ) )
);



                std::cout << "NetVFB Controller running ... (press
<Enter> to exit)" << std::endl;
                std::cin.get();


                return EXIT_SUCCESS;
        }
        catch (NetVFB::Exception& e)
        {
                std::cerr << "Caught exception: " << e << std::endl;
                return EXIT_FAILURE;
        }
        catch (std::exception& e)
        {
                std::cerr << "Caught unhandled exception: " << e.what()
<< std::endl;
```

```
                return EXIT_FAILURE;
        }
}
```

12.3.1.4 *Controller code example (Python, REST)*

The DaaS Controller API is the only one providing an additional REST interface, which can be used from inside all kinds of (typically scripting) languages. Here is an example in Python, adding five Projections of a VFB onto a VD. The Controller used is the one running at the URL given as a command line argument.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from httplib2 import Http
import urllib2
import json
import time
import sys

def __get_displays(controller):
    data = urllib2.urlopen("http://" + controller + "/display/").read()
    return json.loads(data)

def __get_vfbs(controller):
    data = urllib2.urlopen("http://" + controller + "/vfb/").read()
    return json.loads(data)

def __get_projections(controller):
    data = urllib2.urlopen("http://" + controller +
"/projection/").read()
    return json.loads(data)

def main():
    controller = sys.argv[1]

    displays = __get_displays(controller)
    vfbs = __get_vfbs(controller)

    origin = displays[0]['origin']
    vfb = vfbs[0]
```

```
    size = vfb['resolution']
    size['x'] = size['x'] // 5


    h = Http()
    for i in range(5):


        new_p = {   'vfb'       : vfb['url'],
                    'origin'    : origin,
                    'size'      : size,
                    'position'  : {"x":size['x']*i, "y":0,"z":0},
                    'pts'       : ""
                }
        resp, content = h.request("http://" + controller +
"/projection/", "POST",
            json.dumps(new_p).encode())
        print("response:", resp, "content:", content)


        time.sleep(5)


    print "projections", json.dumps(__get_projections(controller),
indent=4, separators=(',',': '))


if __name__ == "__main__":
    main()
```

12.3.2 Application Building and Installation

12.3.2.1 *Building applications using DaaS SDK with CMAKE*

All above-mentioned C++ applications are built using the DaaS SDK. TheSDK uses CMake (version 2.8 or later) to generate the required build system (e.g. Visual Studio).

- Install CMake and DaaS SDK
- Point CMake to the source code ($SDK/examples) and where to build the binaries (preferably out of source)
- Click Configure and select Visual Studio 2010 Win64
- Click Generate to create the solution
- Open the generated solution in the directory where binaries are built
- The Install target automatically deploys the compiled examples to be $SDK/bin folder where all necessary libraries are located

12.3.2.2 *Install APKs on Android*

Applications built for Android (e.g., a VD application using a smartphone's screen real estate), need to be transferred to the device after cross-compiling it in an Android building environment (e.g., Linux and Eclipse). This procedure works as follows.

1. Enable installation of non-market apps on each device:

```
Settings -> Security -> Unknown Sources
```

2. Copy .apk files to device memory, e.g. over USB connection.

3. Locate .apk in any file browser on device.

```
e.g. My Files, Root Browser
```

4. Opening .apk files in file browsers starts the installation process automatically.

13 GIS Data Provider - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

13.1 Introduction

This document describes how to implement web client which is capable to query GIS data from GeoServer in XML3D format. GeoServer is able to create XML3D objects and pass them to the client.

Before starting to implement web client software it is recommended that installation part from the *GIS Data Provider - Installation and Administration Guide* is successfully completed.

13.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the GIS Data Provider GE which is part of the Advanced Middleware and Web User Interfaces chapter. For more background information on this GE, also refer to its Open Specification and Architecture Description.

13.2 User guide

Test GIS client is part of the release. 3D rendering is done with XML3D, therefore web browser needs to support it. XML3D is based on WebGL and JavaScript, any browser that support these two technologies should work. Most tested browsers are:

- Chrome on Windows, Mac OS X and Android
- Firefox on Windows, Mac OS X and Android
- Opera on Android

13.2.1 Setup GeoServer with test data

To be able to query GIS data in XML3D format from GeoServer GIS data itself needs to be uploaded to server. One way to upload test data to GeoServer is by uploading **ESRI Shapefile** to server. This practice was used during implementation of GIS Data Provider GE. Shapefile needs to contain geometry objects with elevation data, otherwise GIS web client is unable to display 3d terrain. Elevation data is used as Z-axis when XML3D objects are generated in GeoServer. When layer as shapefile is uploaded to GeoServer, mandatory filetypes are *.shp, .dbf* and *.shx*.

Compared to data efficiency between data stored to shapefile or PosGIS DB, reading data from PostGIS performs significantly better. Detailed guidance how to setup PostGIS with provided test data is described in the *GIS Data Provider - Unit Testing Plan and Report* -document.

13.2.1.1 *3D GIS data generation*

GIS Data Provider GE test data was based on the National Land Survey of Finland elevation data. Source data was in *-xyz*-format, which contains elevation data with spatial information. With this source information it is possible to generate shapefile consisting polygon structure with elevation data. 2D presentation of the converted shapefile with elevation points is flat grid, in 3D presentation each grid points are in same elevation level as in the real world. Therefore this grid with elevation data can be used as source for terrain presentation.

13.2.1.2 *Uploading XML3D objects reference information to PostGIS*

Single XML3D objects can be placed to 3D GIS environment based on the object location information. Location information can be stored as PointZ presentation to PostGIS.

GeoServer supports reading XML3D reference objects from the PostGIS in following format:

ID	name text	geom geometry(PointZ)	mesh_ref text
1	Cabin1	01010000A0E70B0000295C8F02367D5241B81E854B81F649410000000000208C40	http://reference_to_xml3d_file/xml3d_file.xml
2	Cabin2	01010000A0E70B0000713D0AB7497E5241C3F528DC09E249410000000000208C40	http://reference_to_xml3d_file/xml3d_file.xml
3	Cabin3	01010000A0E70B000048E17A04A888524114AE47A1CEEE49410000000000208C40	http://reference_to_xml3d_file/xml3d_file.xml

Geometry conversion to PostGIS can be done with **psql ST_GeomFromText** -command.

13.3 Programmers guide

Guide for programmers utilizing the GE.

GIS data provider release contains example implementation of the GIS web client (*GISDataProvider-test_client_rel3_3.zip*) which is capable to render terrain data in XML3D format. Note that client implementation is done to support only EPSG:3047 CRS, there's no guarantee support for other CRS.

13.3.1 Needed javascript libraries

For enabling XML3D content in web page at least *xml3d.js* needs to be included.

```
<script src="http://www.xml3d.org/xml3d/script/xml3d.js"></script>
```

If camera handlding is used, include camera.js.

```
<script
src="http://www.xml3d.org/xml3d/script/tools/camera.js"></script>

in order to activate camera moving by keys, useKeys needs to be defined
inside <xml3d>.

<bool id="useKeys">true</bool>
```

Camera handling in 3D GIS environment works following way:

- 'a'- and 'd'-keys move camera left and right
- 'w'- and 's'-keys move camera up and down
- Camera orientation can be changed by pressing right mouse button down and moving mouse

13.3.2 Get GeoServer capabilities

GeoServer capabilities can be queried with following syntax : *http://hostname:port/path?SERVICE=W3DS&ACCEPTVERSIONS=0.3.0,0.4.0&request=GetCapabilities*

The response to a GetCapabilities request is an XML document containing service metadata about the server, including specific information about layer properties and how to access data from the server.

```
Example query from test client:
```

```
http://localhost:9090/geoserver/ows?service=w3ds&version=0.4.0&request=
GetCapabilities
Example of layer details in GetCapabilities response:
 <w3ds:Layer>
   <ows:Title>V4132E</ows:Title>
   <ows:Abstract/>
   <ows:Identifier>fiware:V4132E</ows:Identifier>
   <ows:BoundingBox crs="EPSG:3047">
       <ows:LowerCorner>368000.0 7542000.0</ows:LowerCorner>
       <ows:UpperCorner>374010.0 7548000.0</ows:UpperCorner>
   </ows:BoundingBox>
   <ows:OutputFormat>model/x3d+xml</ows:OutputFormat>
   <ows:OutputFormat>model/xml3d+xml</ows:OutputFormat>
   <ows:OutputFormat>text/html</ows:OutputFormat>
   <w3ds:DefaultCRS>EPSG:3047</w3ds:DefaultCRS>
   <w3ds:Queriable>true</w3ds:Queriable>
   <w3ds:Tiled>false</w3ds:Tiled>
   <w3ds:Style>
       <ows:Title>A default style</ows:Title>
       <ows:Abstract>A sample style that just prints out a green
line</ows:Abstract>
       <ows:Identifier>line</ows:Identifier>
   <w3ds:IsDefault>true</w3ds:IsDefault>
   </w3ds:Style>
 </w3ds:Layer>
```

- **<ows:Identifier>** tag contains layer name which can be used for querying specific layer data.
- **<ows:BoundingBox>** contains information of the total bounding box area where layer data is located. Layer data can be requested inside total bounding box. There is also information which Coordinate Reference System (CRS) layer uses.
- **<model/xml3d+xml>** indicates that layer output can be requested as in XML3D format.
- **<w3ds:Queriable>** states if layer is queriable for client. In case queriable value is false client is not able to request layer data.

13.3.3 Query XML3D objects from GeoServer

important First of all it is important to explain how GeoServer returns XML3D data: XML3D GIS data queries are always done with real spatial location with in CRS which layer supports. Layer queries needs to contain at least *W3DS service definition, layer CRS, layer name and layer bounding box*. **GeoServer will** process data query and **return result always located to origin (0,0)**. For this reason client needs to be aware internally which part of the whole layer area is drawn and to where GIS camera is directed. In a short client needs to implement internal scene manager. Without scene manager all layers returned by GeoServer are put to same origin.

XML3D object query can be done based on the GetCapabilities response data. XML3D layer which has *<ows:OutputFormat>model/xml3d+xml</ows:OutputFormat>* and *<w3ds:Queriable>true</w3ds:Queriable>* can be queried from the server. It is advised that layer queries should be done with in layer bounding box. There's no actual harm to extend query outside of the bounding box, just no data to return from that area.

```
XML3D object query syntax:
```

```
geoserver/w3ds?version="version"&service=w3ds&request="request
type"&crs="layer CRS"&format="format for XML3D response"&layers="layer
name"&boundingbox="query area for GIS data"

Example how the partial layer data is queried, whole layer area is
248000,7668000 260010,7680000.


localhost:9090/geoserver/w3ds?version=0.4&service=w3ds&request=GetScene
&crs=EPSG:3047&format=model/xml3d+xml&layers=fiware:saana&boundingbox=2
48000,7668000,252003,7672000
```

GeoServer returns XML3D object definitions or references to XML3D definition files.

13.3.3.1 *Level Of Details (LOD) usage in object query*

Level Of Details (LOD) is integer value starting from 1 and ending to 10. Smaller than 10 LOD levels are generated so that original source data is filtered in GeoServer based on LOD level so that generated 3D terrain data has less details compared to original. LOD level 10 means that detail levels are not reduced at all from the source data.

Level Of Detail is defined in the GeoServer query by providing *LOD* -parameter with relevant LOD number.

```
Example how to add define LOD level 4 (&LOD=4) in the GeoServer query


130.206.80.182:8080/geoserver/w3ds?version=0.4&service=w3ds&request=Get
Scene&crs=EPSG:3047&format=model/xml3d+xml&layers=testbed:fiware_test_t
errain&boundingbox=373969.9375,7547970,375183.9625,7549182&LOD=4
```

13.3.3.2 *Returned XML3D data object definition*

When Geoserver returns XML3D object definition it contains following information:

```
<group xmlns="http://www.w3.org/1999/xhtml"
id="outputGeometryCollection" class="NodeClassName">
  <mesh type="triangles">
    <int name="index">...</int>
    <float3 name="position">...</float3>
    <float2 name="texcoord">...</float2></mesh></group>
```

Above information is according to XML3D API specification and it needs to be placed inside <xml3d>...</xml3d> -tags. Web client needs to create light shader for the loaded XML3D content.

13.3.3.3 *Adding texture*

Texture for returned XML3D terrain object can be added by requesting graphics with same bounding box as requested terrain XML3D model was requested.

```
Texture query syntax:

geoserver/w3ds?version="version"&service=w3ds&request="request
type"&crs="layer CRS"&format="format for XML3D response"&layers="layer
name"&boundingbox="query area for texture data"

geoserver/wms?service=WMS&version=1.1.0&request=GetMap&layers="texture
layer name"&styles=&bbox="query area for texture"&width="texture
width"&height="texture height"&srs="texture layer crs"&format="image
format"
```

```
Example for requesting terrain texture:

dev.cyberlightning.com:9090/geoserver/fiware/wms?service=WMS&version=1.
1.0&request=GetMap&layers=fiware:NorthernFinland_texture&styles=&bbox=3
56000,7530000,372003.3333333333,7546000&width=1024&height=1024&srs=EPSG
:404000&format=image%2Fpng
```

One option to store terrain texture to GeoServer is by using GeoTIFF. In this case Layer bounding box needs to be precisely correct so that GeoServer is able to return correct area of the bitmap when requested.

13.3.3.4 *Returned XML3D data definition file reference*

GeoServer is capable to include external XML3D object definition references in to the returned XML3D GIS data query. Chapter *Uploading XML3D objects reference information to PostGIS* describes how data should be uploaded to PostGIS. GeoServer will query database based on the requested bounding box and returns all found points. Each point has reference to external XML3D file. Each point are translated to coordinates inside requested bounding box, so client scene manager needs to be aware where returned building objects should be placed in the 3D GIS presentation.

Example of returned XML3D information with external reference:

```
<group xmlns="http://www.w3.org/1999/xhtml"
id="fiware:building_coordinates" class="" style="transform:
translate3d(1102,900.0,-0.0)">

 <mesh src="http://localhost:8989/a416a634c021.json "></mesh>

</group>
```

13.3.4 Add XML3D objects to html page

XML3D objects can be injected to client web page or generate new web page where returned XML3D objects are inserted. XML3D declarations received from the GeoServer needs to be injected inside <xml3d> -tags.

Example of index.html-template where new XML3D objects are injected. In GIS data provider demo all XML3D elements are injected inside *id="MaxScene"* -group.

index.html before XML3D data request:

```
 <body>
  <div>
    <xml3d id="xml3dContent" xmlns="http://www.xml3d.org/2009/xml3d">
        <defs id="defs" xmlns="http://www.xml3d.org/2009/xml3d">
            <transform id="t_node-camera_player" rotation="0.0 0.0 0.0
0.0" translation="0 0 0"/>
            <lightshader id="light1"
script="urn:xml3d:lightshader:directional" >
              <float3 name="intensity" >2 2 2</float3>
              <bool name="castShadow">true</bool>
              <float3 name="direction">1 -0.5 1.0</float3>
            </lightshader>
```

```
            <light shader="#light1" />
            <bool id="useKeys">true</bool>
        </defs>

        <group xmlns="http://www.xml3d.org/2009/xml3d" id="MaxScene">
          <group id="node-camera_player" transform="#t_node-
camera_player" lightshader="#light1">
            <view fieldOfView="0.7" id="camera_player-camera"/>
          </group>
        </group>
    </xml3d>
  </div>
 </body>
```

index.html after XML3D data request:

```
 <body>
  <div>
    <xml3d id="xml3dContent" xmlns="http://www.xml3d.org/2009/xml3d">
        <defs id="defs" xmlns="http://www.xml3d.org/2009/xml3d">
            <transform id="t_node-camera_player" rotation="0.0 0.0 0.0
0.0" translation="0 0 0"/>
            <lightshader id="light1"
script="urn:xml3d:lightshader:directional" >
              <float3 name="intensity" >2 2 2</float3>
              <bool name="castShadow">true</bool>
              <float3 name="direction">1 -0.5 1.0</float3>
            </lightshader>

            <light shader="#light1" />
            <bool id="useKeys">true</bool>
        </defs>

        <group xmlns="http://www.xml3d.org/2009/xml3d" id="MaxScene">
          <group id="node-camera_player" transform="#t_node-
camera_player" lightshader="#light1">
            <view fieldOfView="0.7" id="camera_player-camera"/>
          </group>
          <group xmlns="http://www.w3.org/1999/xhtml"
id="fiware:building_coordinates" class="" style="transform:
translate3d(1102,900.0,-0.0)">
```

```
            <mesh src="http://localhost:8989/a416a634c021.json
"></mesh>
          </group>


        </group>
    </xml3d>
  </div>
 </body>
```

14 POI Data Provider - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

14.1 Introduction

This document describes how to design and implement a distributed system utilizing the POI Data Provider Open Specification.

The available architecture gives wide possibilities to develop systems utilizing location data. Publicly available generic data can be augmented by business or application specific data. The system can utilize private data with restricted access.

Considerations in distributed system architecture are covered in System design considerations.

Server design principles are covered in Server programming guide.

Client design and implementation is covered in Client programming guide.

14.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the POI Data Provider GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

14.2 User guide

This section does not apply as this GE implementation is for programmers, who will invoke the APIs programmatically, there is no user interface as such.

14.3 Programmers guide

14.3.1 System design considerations

System architecture based on POI servers is quite free. It is possible to use publicly available and proprietary POI data providers together.

14.3.1.1 *Using distributed data*

A POI related application may have needs exceeding the capabilities of available public POI data providers, e.g.:

- More data must be associated to POIs
- More POIs are needed
- Private extensions to POIs and data are needed
- High-integrity data are needed

These needs can be addressed using several POI databases together. Private POI servers with possible access control can extend the scope of this technology to **demanding critical solutions**.
It is easy to **combine data in several POI servers** for use.

- A client program can query several POI servers.
- A POI server can query other POI servers.

A query may request

- more POIs and

- more data on given POIs.

Because POIs are identified by UUIDs it is possible to combine data about a POI from several otherwise unrelated POI data providers.

The UUID of a POI must be the same in different databases, or explicit mapping is needed.

14.3.1.2 *Using separate data*

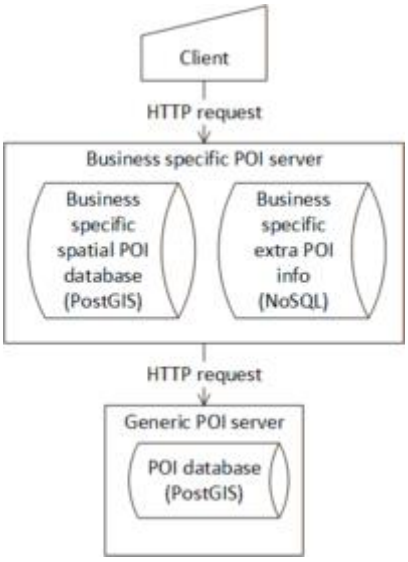
Of course, public POI data providers are not required, if they are not useful for the application. The application can use POI data provider(s) and POIs totally separated from the publicly available ones.

14.3.1.3 *Using SQL and NoSQL databases*

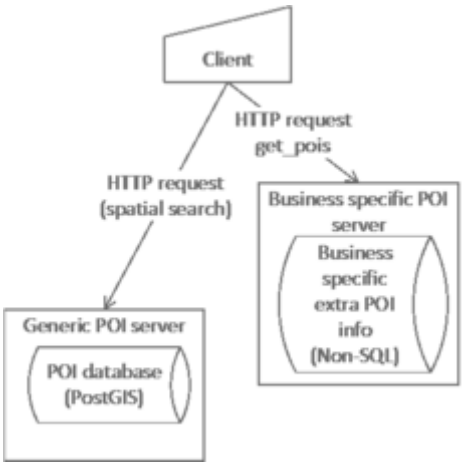
SQL databases, including PostGIS, are good in searches with several limiting conditions. However, they are laborious to program for given data content. NoSQL databases are easy to use for arbitrary data, but they are worse in general searches. This POI architecture uses PostGIS database for searches based on spatial and other conditions. NoSQL database is used to store other information about POIs. PostGIS database provides the UUID of the POI. The UUID is used as a key to access a NoSQL database for the rest of the data.

14.3.1.4 *Architecture examples*

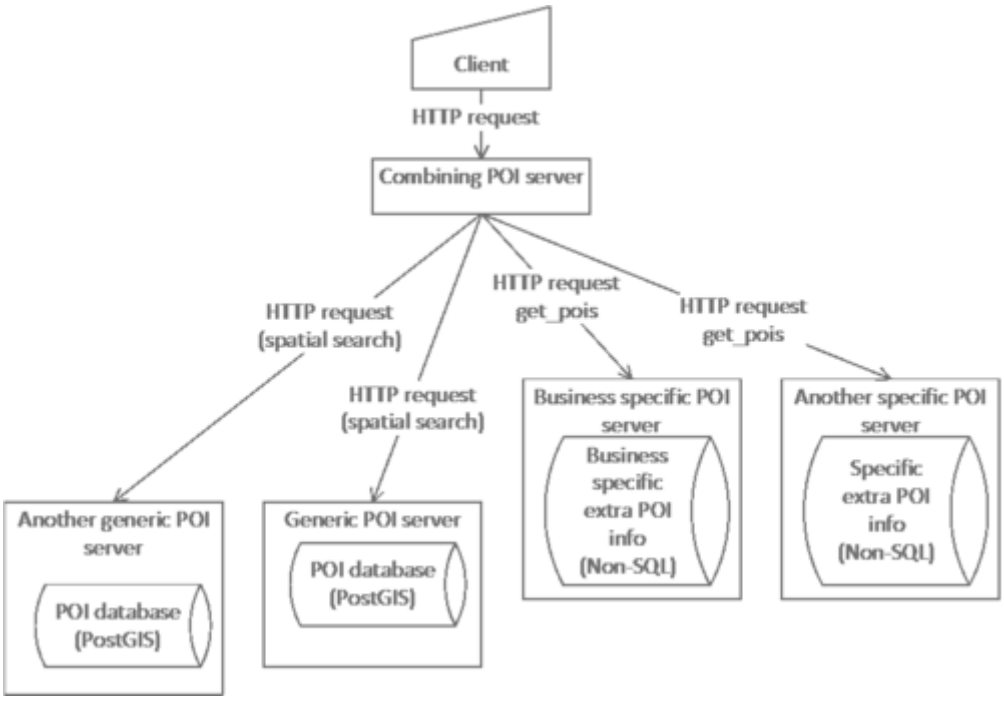
Following are some configurations combining several POI data providers.



Business specific POI server uses another POI server



Client uses several POI servers



General case: A POI server combines data from several POI data providers

14.3.2 Interface reference

Interfacing between client and server is defined in POI Data Provider Open API Specification.

14.3.3 Server programming guide

14.3.3.1 *Implementing a special-purpose POI Data Provider*

You can use the provided POI Data Provider implementation as a basis for writing your own specific implementation. The part of the code you most likely have to modify the most are the database queries, as you may have specified your own database model. The following links contain guides for using both PostGIS and MongoDB:

Introduction to PostGIS

The MongoDB 2.4 Manual

14.3.3.2 *Composing a POI Data Provider from multiple backends*

You can write a POI Data Provider backend that composes data from multiple different POI Data Provider backends. The high-level application logic for such a implementation is presented here:

Handling spatial queries (e.g. radial_search)

1. Forward the spatial query to a POI backend that is capable of handling them (core backend)
2. Parse the response to the spatial query and create a list of POI UUIDs
3. Loop through the rest of the POI backends and request additional data components for each UUID with get_pois request
4. Construct the final output JSON by appending the additional data components for each POI

Handling get_pois queries

The get_pois query is a simpler case than the spatial queries, as you directly get the list of UUIDs as the request parameter:

1. Loop through the POI backends and request additional data components for each UUID with get_pois request
2. Construct the final output JSON by appending the additional data components for each POI

14.3.4 Client programming guide

See Interface reference for

- **details of communication** with POI servers and
- essential **data structures** provided by POI servers.

14.3.4.1 *General on client design*

General pattern of client operation is

1. sorting out the area and other attributes of current interest - possible user interaction,
2. **generic query** for POIs using spatial and other conditions,
3. selecting the POI(s) to be focused on - possible user interaction,
4. **specific query** for more data on specific POIs,
5. using (showing) the acquired POI data - possible user interaction.

This manual describes how to program the queries to POI servers. Language used in examples is JavaScript due its wide availability in web browsers.

The common parts used in queries are described in separate section.

XMLHttpRequest is used to perform REST queries.

14.3.4.2 *Spatial query*

Spatial query is used to find the POIs based on their location. See Interface reference for complete treatment of available query choices.

JavaScript skeleton for requesting POIs in given radius below gives an example how to implement a query in a client.

```
 /*
   Query parameters:


   BACKEND_ADDRESS_POI - example: "http://dev.cie.fi/FI-WARE/poi_dp/"
   lat - latitude of the center point, degrees north
   lng - longitude of the center point, degrees east
   searchRadius - meters
   languages - a string array containing the accepted language codes
 */
 var query_url = BACKEND_ADDRESS_POI + "radial_search?" +
  "lat=" + lat + "&lon=" + lng + "&radius=" + searchRadius +
"&component=fw_core";

 poi_xhr = new XMLHttpRequest();
 poi_xhr.onreadystatechange = function () {
   if(poi_xhr.readyState === 4) {
     if(poi_xhr.status  === 200) {
```

```
      var resp_data = JSON.parse(poi_xhr.responseText);
      process_response(resp_data);
    }
    else {
      console.log("failed: " + poi_xhr.responseText);
    }
  }
}

poi_xhr.onerror = function (e) {
  log("failed to get POIs");
};

poi_xhr.open("GET", query_url, true);
set_accept_languages(poi_xhr, languages);
poi_xhr.send();
```

14.3.4.3 *Additional data query*

Additional data for POIs already found can be requested using UUIDs of POIs as keys. Below is a JavaScript skeleton for requesting more data on given POIs.

```
/*
  Query parameters:


  BACKEND_ADDRESS_X_POI - example: "http://dev.cie.fi/FI-WARE/poi_dp/"
  uuids[] - string array containing UUIDs of interesting POIs
  languages - a string array containing the accepted language codes
*/
var query_url = BACKEND_ADDRESS_X_POI + "get_pois?poi_id=" +
join_strings(uuids, ",");

poi_xhr = new XMLHttpRequest();
poi_xhr.onreadystatechange = function () {
  if(poi_xhr.readyState === 4) {
    if(poi_xhr.status  === 200) {
      var resp_data = JSON.parse(poi_xhr.responseText);
      process_response(resp_data);
    }
    else {
      console.log("failed: " + poi_xhr.responseText);
    }
  }
}

poi_xhr.onerror = function (e) {
  log("failed to get POIs");
};

poi_xhr.open("GET", query_url, true);
set_accept_languages(poi_xhr, languages);
poi_xhr.send();
```

14.3.4.4 *Handling received POI data*

Query response consists of `pois` data on requested POIs, which consists of POI items having POI UUIDs as their keys.

Example data (shortened and annotated):

```
{
  "pois": {
    "6be4752b-fe6f-4c3a-98c1-13e5ccf01721": {
      "fw_core": {
        "category": "cafe",
        "location": {<location of Aulakahvila>},
        "name": {
          "": "Aulakahvila"
        },
        <more core data on Aulakahvila>
      },
      <more data components on Aulakahvila>
    },
    "ae01d34a-d0c1-4134-9107-71814b4805af": {<data on restaurant Julinia>},
    "1c022820-62dc-487b-95b4-6c344d6ba85e": {<data on library
Tiedekirjasto Pegasus>},
    <more data on more POIs>
  }
}
```

First, the received text is parsed using **JSON.parse()** function.
NOTE: For security reasons do not use `eval()` function to parse the data! Maliciously formatted data can be used to compromise the security.

Then, the resulting data structure is processed for POI data. The function skeleton `process_response` below is an example how to process the data.

For the detailed structure of the response data see Interface reference.

14.3.4.4.1 *`process_response( data )` - skeleton*

```
function process_response( data ) {

    var counter = 0, jsonData, poiData, pos, i, uuid, pois,
        contents, locations, location, searchPoint, poiCore,
        poiXxx;

    if (!(data && data.pois)) {
        return;
    }

    pois = data['pois'];

    /* process pois */

    for ( uuid in pois ) {
```

```
            poiData = pois[uuid];
            /*
               process the components of the POI
               e.g. fw_core component containing category, name,
               location etc.
               Taking local copies of the data can speed up later
               processing.
            */
            poiCore = poiData.fw_core;
            if (poiCore) {
              /* fw_core data is used here */

            }
            /* Possible other components */
            poiXxx = poiData.xxx;
            if (poiXxx) {
               /* xxx data is used here */

            }
        }
    }
```

14.3.4.5 *Adding a new POI*

Below is a JavaScript skeleton for adding a new POI to the POI-DP.

```
    BACKEND_ADDRESS_POI = "http://dev.cie.fi/FI-WARE/poi_dp/";

    poi_data = {fw_core: {...},  -other components- };

    function addPOI( poi_data ) {
        var restQueryURL;

        restQueryURL = BACKEND_ADDRESS_POI + "add_poi";
        miwi_poi_xhr = new XMLHttpRequest();

        miwi_poi_xhr.overrideMimeType("application/json");

        miwi_poi_xhr.onreadystatechange = function () {
            if(miwi_poi_xhr.readyState === 4) {
                if(miwi_poi_xhr.status  === 200) {
                  // React to successfull creation
                  //  alert( "success: " + miwi_poi_xhr.responseText);
```

```
                }
                else {
                  // React to failure
                  // alert("failed: " + miwi_poi_xhr.readyState + " " +
miwi_poi_xhr.responseText);
                }
            }
        }
        miwi_poi_xhr.onerror = function (e) {
            // React to error
            // alert("error" + JSON.stringify(e));
        };
        miwi_poi_xhr.open("POST", restQueryURL, true);
        miwi_poi_xhr.send(JSON.stringify(poi_data));
    }
```

14.3.4.6 *Updating POI*

Below is a JavaScript skeleton for updating POI data in the POI-DP.

First, the data is fetched from the server.

```
    BACKEND_ADDRESS_POI = "http://dev.cie.fi/FI-WARE/poi_dp/";


    var query_handle;


    function POI_edit(uuid) {
    /*
        uuid - the id of the POI to be updated
    */
        var restQueryURL, poi_data, poi_core;
        // get_for_update brings all language variants etc.
        restQueryURL = BACKEND_ADDRESS_POI + "get_pois?poi_id=" + uuid +
            "&get_for_update=true";

        console.log("3D restQueryURL: " + restQueryURL);
        query_handle = new XMLHttpRequest();

        query_handle.onreadystatechange = function () {
            if(query_handle.readyState === 4) {
                if(query_handle.status  === 200) {
                    //console.log( "succes: " + xhr.responseText);
                    var json = JSON.parse(miwi_3d_xhr.responseText);
                    var poi_edit_buffer = json.pois[uuid];

                    /* Here a data editor is opened to edit the contents
of poi_edit_buffer.
                       updatePOI is a callback that is called to send
update to server.
                       uuid is passed to the callback. Other parameters
are for information, only.
                    */
```

```
                    A_nonspecific_data_editor("update poi "  + uuid,
"Edit POI data", poi_edit_buffer, uuid, updatePOI);

                }
            }
        }
        query_handle.onerror = function (e) {
            log("failed to get data");
        };

        query_handle.open("GET", restQueryURL, true);
        query_handle.send();
    }
```

When editing is ready, the new version of data is sent to the server.

```
function updatePOI( poi_data, uuid ) {
/*
  poi_data is the updated version of POI data like:
    {
      "fw_core": {...},
      "fw_times": {...}
    }
  uuid is the id of the POI
*/
    var restQueryURL;
    var updating_data = {};
/* build updating structure like
    {
      "30ddf703-59f5-4448-8918-0f625a7e1122": {
        "fw_core": {...},
        ...
      }
    }
*/
  updating_data[uuid] = poi_data;


  restQueryURL = BACKEND_ADDRESS_POI + "update_poi";
  query_handle = new XMLHttpRequest();

  query_handle.overrideMimeType("application/json");

  query_handle.onreadystatechange = function () {
      if(query_handle.readyState === 4) {
          if(query_handle.status  === 200) {
```

```
            // Here we may notify the user of successfull update
            // alert( "success: " +query_handle.responseText);
          }
      }
   }
   query_handle.onerror = function (e) {
       // Something bad happened
       alert("error" + JSON.stringify(e));
   };
   // define the operation and URL
   query_handle.open("POST", restQueryURL, true);
   // send the data
   query_handle.send(JSON.stringify(updating_data));
 }
```

14.3.4.7 *Deleting POI*

Below is a JavaScript skeleton for deleting a POI from the POI-DP.

```
   BACKEND_ADDRESS_POI = "http://dev.cie.fi/FI-WARE/poi_dp/"; // for
example


   function POI_delete(uuid) {
     var restQueryURL, poi_data, poi_core;


     var cfm = confirm("Confirm to delete POI " + uuid);
     if (cfm)
     {
       // build the URL for delete
       restQueryURL = BACKEND_ADDRESS_POI + "delete_poi?poi_id=" + uuid;

       miwi_3d_xhr = new XMLHttpRequest();
       // populate the request with event handlers
       miwi_3d_xhr.onreadystatechange = function () {
           if(miwi_3d_xhr.readyState === 4) {
               if(miwi_3d_xhr.status  === 200) {
                   // Notify user about success (if wanted)
                   alert("Success: " + miwi_3d_xhr.responseText);
               }
           }
       }

       miwi_3d_xhr.onerror = function (e) {
           log("failed to delete POI " + JSON.stringify(e));
       };

       // Note: It seems to help DELETE if the client page is in the
       //       same server as the backend
       miwi_3d_xhr.open("DELETE", restQueryURL, true);
       miwi_3d_xhr.send();
     }
  }
```

14.3.4.8 *Utility functions*

14.3.4.8.1 *join_strings(strings_in, separator)*

This function can be used to make a comma separated string from an array of strings.

```
function join_strings(strings_in, separator) { //: string
  /*
    strings_in string array
    separator string to be inserted between strings_in

    *result string - strings of strings_in separated by separator

    Example: join_strings(["ab", "cd", "ef"], ",") -> "ab,cd,ef"
  */
  var result, i;

  result = strings_in[0] || "";
  for (i = 1; i < strings_in.length; i++) {
    result = result + separator + strings_in[i];
  }

  return result;
}
```

14.3.4.8.2 *set_accept_languages(http_request, languages)*

This function is used to define preferred languages for query responses. The language preferences are coded to the `Accept-Languages` header of the http request.

```
set_accept_languages(http_request, languages) {
  /*
    This function creates an Accept-Languages header to the HTTP
request.
    This must be called between http_request.open() and
    http_request.send() .

    http_request - an instance of XMLHttpRequest
    languages    - string array containing the codes of the languages
                   accepted in the response in descending priority.
                   The ISO 639-1 language codes are used. If any
language
                   texts are accepted in case of none of the listed
                   languages are found, an asterisk is used as the
last
```

```
                    code.
                    Example: ["en","fi","de","es","*"]
   */
       var i, q;

       q = 9;
       for (i = 0; i < languages.length; i++) {
         if (i == 0) {
           http_request.setRequestHeader('Accept-Language',
languages[0]);
         } else {
           if (languages[i] != "") {
             http_request.setRequestHeader('Accept-Language',
languages[i] +
                 ';q=0.' + q);
             if (q > 1) {
               q--;
             }
           }
         }
       }
 }
```

15 2D-3D Capture - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

15.1 Introduction

2D-3D capture GE is a setting that consists of a two components.

- Device Web API.
- Rest API for Python server.

Device Web API interfaces between web applications and web browsers device API. Web Browser Device API is comprehensive API which provide access to hardware elements and information about hardware to web applications. Current implementation of Device Web API brings an abstraction layer to web applications where user is able to access some of the sensors that are included in the Browser Device API in a browser independent manner.Second task of Web API is to provides an interface to web applications to access Rest server running on a public server which facilitates uploading multimedia content to tag with metadata.

REST server which is the back bone of the GE is the storage unit. It also provides a REST API for users to access and manipulate images and related metadata to be used in 2D-3D contexts and also provide means for other GE's to access video and image (Visual information) from the repository.

In order to cover both the aspects User Guide and the Programmers Guide consists of two sections.

This User and Programmers Guide relates to the 2D-3D capture GE which is part of the Advanced Middleware and Web User Interfaces chapter. For more background information on this GE, also refer to its Open Specification and Architecture Description.

15.2 User guide

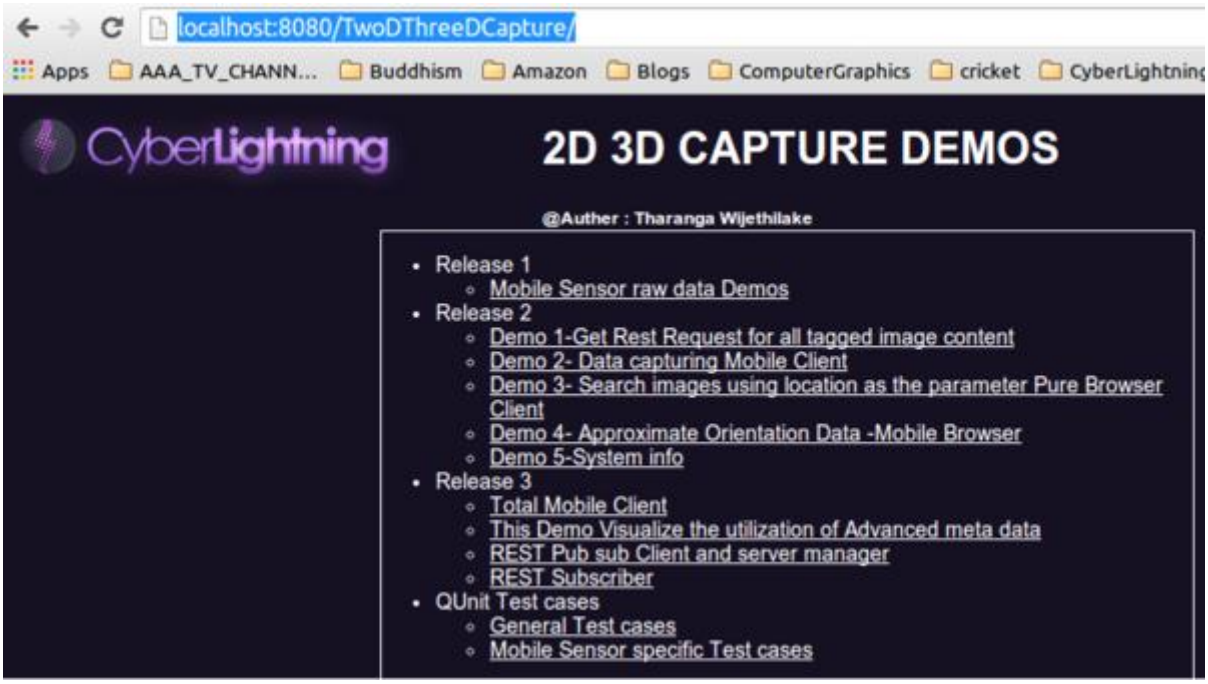
User guide of 2D-3D capture components describes how to run the web clients developed using the REST Server and the web API. User guide assumes the server is running on the ports that are described in the installation and administration guide and web components are accessible from public domain.

15.2.1 Running web demos

This part of the document assumes that the Rest server to be running on a public server and the web client to be deployed at a accessible server. All the demos require WegGL supporting web browsers. Once the web client is deployed it is accessible from following URL. Default port for Tomcat in the settings file is 9090, but but the port in the URL should be changed to the port which tomcat is running. Following figure shows application running on a localhost running on port 8080.

```
http://www.example.com:9090/TwoDThreeDCapture
```

This will pop up the following page.



This list of applications includes a list of web applications developed using the the java script web API developed in the GE. This document focuses on the links indicated under the topic "Release 3"

- Total Mobile Client - 2D-3D capture Mobile demo web

This is the web client developed using the 2D-3D capture web API. Page is accessible at

`http://www.example.com:9090/TwoDThreeDCapture/Camerafeed.html`

- Advanced Meta data visualization.

This demo is intended to run on desktop computer browsers(Firefox, Chrome.) This demonstration indicates few images taken from mobile client that are ordered according to sensor data collected from mobile device.

- Rest Pub-Sub server and sanity check client.

This client is to prepare for the image uploads. In order for the next demo to function properly this page should be used to open prepare the server. In order to use this test page python server component needs to be run on publicly accessible host.

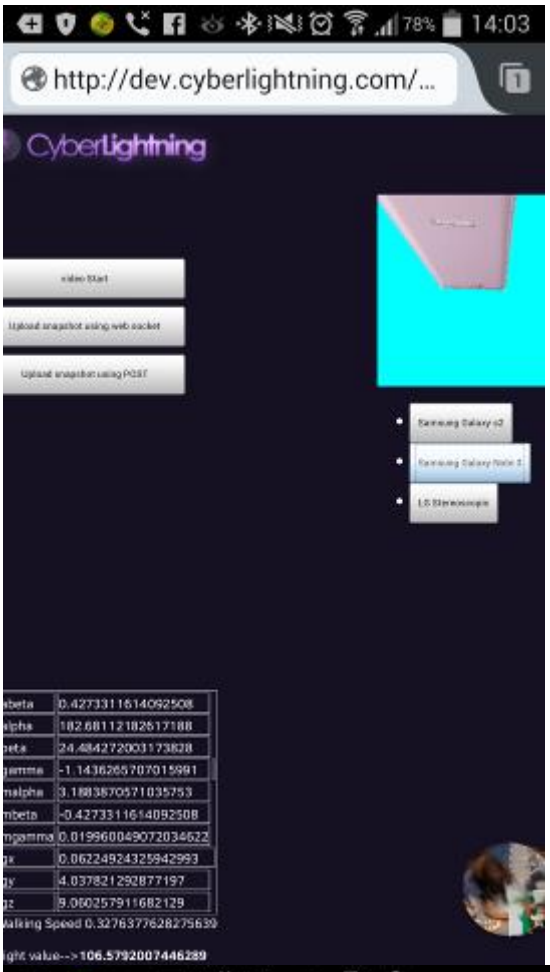
- Rest subscriber.

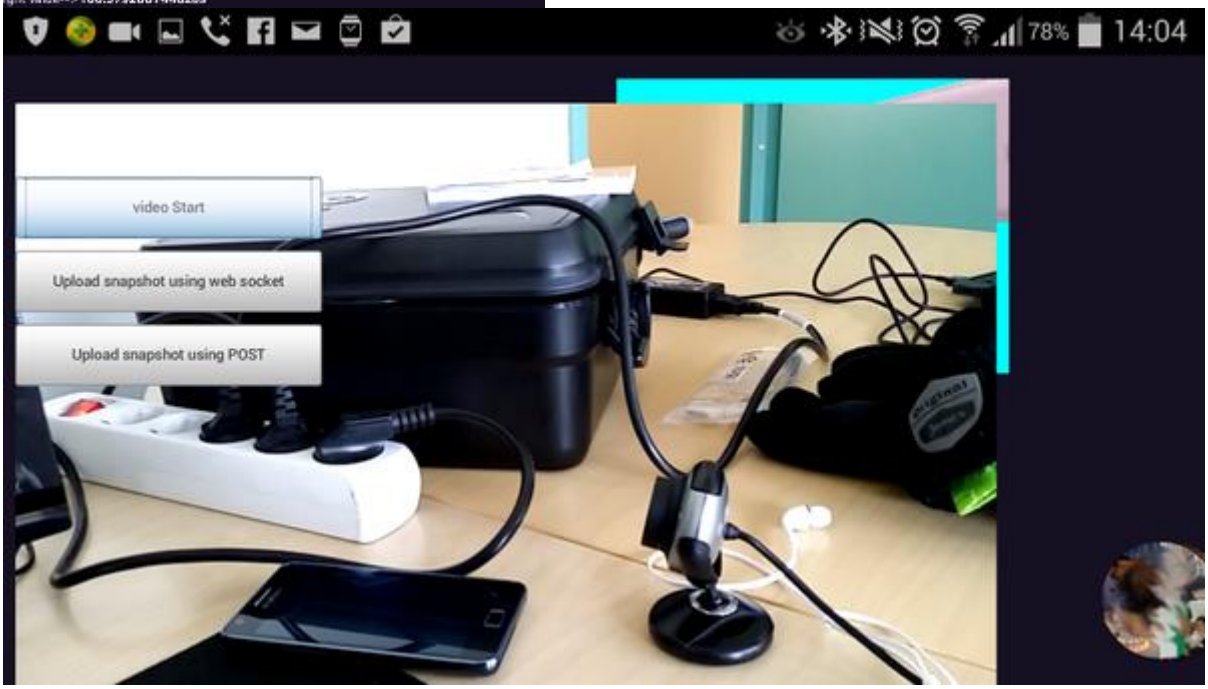
This is the desktop client of a live demo that shows uploaded images on right locations in right orientations. This works in synchronization with the Total mobile client and the Back end server. Further this service depends on GIS service and will provide map information for images taken in Oulu Area. This demo assumes both the python server and the web server this client is running are deployed on the server server. if not user need to edit the DesktopSubscriptClient_Demo1.js and ImageOrientationClient.js to point to the correct server. in both the files following line need to edit to point to the correct server. Following section describes the steps to run the demo.

15.2.1.1.1 *Running the Demo to visualize Utilizing advanced metadata on webGL context*

1. Open the Third link"REST Pub sub Client and server manager" on under "Release 3-". Press "Start Server". When a response message arrived from the server there is a popup message that indicates the event completion. Then press "Init" and wait till the response message arrive. It is a must that in both occasions user should await response from the server.
2. Open a new tab on the browser and open the fourth link "REST Subscriber".

3. Press "connect to server" which will show a pop up ,when the server received the contact request. Then press "Press-To-Receive" and client is acknowledged accordingly. WebGL context will open a blue window and now ready to receive events from the server with relevent to any image upload.
4. Open the "Total Mobile Client "web page mobile client on the mobile device. Press "Start Video" and then press either of the next buttons ("Upload snapshot using web socket" or "Upload snapshot using POST"). Following images show the relevant views on a mobile display. Once the image upload is completed it will be shown on the display in the correct orientation.





This mobile web client depends on the sensor data collected from mobile device. This sensor data is Samsung galaxy s2 and Samsung galaxy Note 3. API is not calibrated for other devices may not work properly on other devices. On the mobile display on the upper right hand side the orientation of the mobile is displayed using the accelerometer data.

15.3 Programmers guide

15.3.1 Web API for mobile web applications

This section describes how to use the js_api.js, the mobile web API designed to assist developers to obtain data from the sensors of a mobile device and to use them in web applications. This API provides access to direct information feeds from the sensors as well as calculated information such as velocity and orientation information of the device. Demo applications that demonstrate the functionality are included in the source package below. These demo applications can be accessed by deploying them in a public Tomcat server. Once deployed application is accessible at

```
http://<path to tomcat server>:<port>/TwoDThreeDCapture.
```

Mobile device is needed to run the mobile specific web pages.

Source code for this web component and its demos can be found at Web Client.

Users of Device WebAPI are programmers and rest of the document assumes the knowledge on HTML and JavaScript.

- This API depends on jquery 2.0.3. Hence step 1 is to insert jquery to the project HTML file.

```
<script type="text/javascript" src="jquery-2.0.3.js"></script>
```

- Insert the javascript js_api.js in the main HTML file.

```
<script type="text/javascript" src="js_api_r2.js"></script>
```

- Create a new javascript source file and import the file to HTML file.

```
<script type="text/javascript" src="Demo2.js"></script>
```

Example HTML file looks like the once given below. It consists of necessary elements(Video) including the live video feed from the device camera.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script type="text/javascript" src="jquery-2.0.3.js"></script>
<script type="text/javascript" src="js_api_r2.js"></script>
<script type="text/javascript" src="Demo2.js"></script>
<link rel="stylesheet" href="camerafeed.css">
</head>
<body>
    <video src="" autoplay id="videopad"></video>
</body>
</html>
```

Once the HTML file is in place next step is to add some content to Demo2.js file. Following steps creates a device object activate available sensors and tries to obtain data from the sensors to a web application.

- Create a Device Object

```
   var dAPI = new FIware_wp13.Device(Local_Proxy_URL
,Remote_Rest_Server_URL, Local_Proxy_server_port,
Web_Socket_Port,Rest_Server_Port)
```

- Local_Proxy_URL : localhost and by default set to "Localhost"
- Remote_Rest_Server_URL :Server host name where the Python flask server is deployed
- Local_Proxy_server_port : if a Local proxy is used to avoid cross domain errors this parameter can be used. In current API this is not necessary.
- Web_Socket_Port : Port that the cleint web sockets connect to upload images
- Rest_Server_Port : Port that the REST services are available.

Device object is the main API function and it is a **MUST TO HAVE** to use the API. Current implementation assumes using a proxy in order to avoid AJAX domain specific issues and parm1 is the URL of the local server. In a test environement this could be localhost but in a mobile application this should be in the same domain as the web page. Remote Rest server is the URL which the mobile application coupled to.Local Proxy server port and Rest server port are TCP ports that are used in servers. In Demo applications proxy server is deployed as a Java Server as a Servlet so the port is the Tomcat port. By default this is 8080 and the Rest server is set to run on port 17324. Rest Server consists of 2 listening ports the second one being the web socket port which enables applications to use web socket connection of the JS_API.Parm 4 is this web socket port.

Following are the main API functions provided by API.

```
   * Following  code lines enable to obtain values from respective
sensors.

   dAPI.showVideo(testCallback)

   function testCallback() {} //Callback function on a success of
obtaining the video feed to the specified video object

   dAPI.getCurrentLocation(onLocationSearchSuccess);

   function onLocationSearchSuccess(pos){}
```

pos is of the type Coordinates

```
   dAPI.registerForDeviceMovements(onLocationSearchSuccess,
onLocationServiceSearchError);

   function onLocationSearchSuccess(pos,coords){}

   function onLocationServiceSearchError(){}
```

pos consist of direct values from the sensor,intention of the Coordinates is the give access to given number of past GPS values. Current setting holds past most recent 20 GPS location values.

```
   dAPI.registerDeviceMotionEvents(handlacceleration,
handleAccelerationWithGravityEvent, handleRotation);

   function handlacceleration(a){}

   function handleAccelerationWithGravityEvent(accelerationWithGravity)
{

       accelerationWithGravity.x

       accelerationWithGravity.y

       accelerationWithGravity.z
```

```
    };
    function handleRotation(r) {
        r.alpha
        r.beta
        r.gamma
    }
```

This functions offer access to values from accelerometer. This values are checked for possible errors in sign(positive negative) and erroneous values are removed and averaged to increase accuracy. Current implementation considered 10 values from immediate past to average. Rotation is not handled but raw values are passed to the call back function.

```
    dAPI.reigsterDeviceOrentationEvent(handleOrientationChanges);
    function handleOrientationChanges(alpha, gamma, beta ,
orientation){}
```

explanation of alpha, beta and gamma values are in this document. Orientation stands for portrait or landscape modes.

- Enable Debugger for debugging options. Once you have this it is possible to print out the values to the logger as follows.

```
    dAPI.setupLogger();
```

- This API is tested on Firefox and Chrome on LG and Samsung android devices. Desktop and Laptop camputers may have trouble using sensor values as they do nto comprise of necessary sensors. Minimal setting requires a GPS sensor and device compass, accellerometer are supported

15.3.2 Rest API

Source code for this web component and its demos can be found at RestServerAPI

Rest API is an python server which supports two basic functions.

- Obtain and store information from mobile browser application.
- Provide information on demand for external services.

- Rest API supports Direct posting images with metadata in the following format as an binary array and also it accepts websocket messages in the same format.

```
    http://www.example.com:PORT/postBinaryImage   POST


    1.First three digits explain the length of the metadata.eg 345.
    2.Metadata is of the following format
    message {
        "type":"image",                 // Image/Video
        "time":"2013.10.10_9.44.1",
//year_month_day_hours_minutes_Seconds This is used as the Image name
        "ext":"jpg",                    //Image compression type. JPG and
PNG are supported
        "position":{
            "lon":30.402053,            //Longiture
```

```
            "lat":62.278423999999994, //Latitude
            "alt":null,                //Altitude
            "acc":122000               //Accuracy
       },
       "motion":{
            "heading":null,           //Heading Direction
            "speed":null              //Speed
        },
        "device":{                    //Seonser information
            "ax":400,"ay":400,"az":400,"gx":400,"gy":400,"gz":400,
//Accellerometer and compass information
            "ra":400,"rb":-181,"rg":-181,
//Compass values.
            "orientation":"potrait"
        },
        "dTime":1381387440723,
        "Device OS" : XXXX,
        "Device Type":XXX,
        "Video High :1234,
        "Video width" :234
    }
```

This structure is the same as the one given below.

```
    3. Image data posted as RGBA data.
```

- Following RestFunction provides a links to all the images and their meta data. It provides GPS information, orentation in the form of Pitch, Roll and Yaw

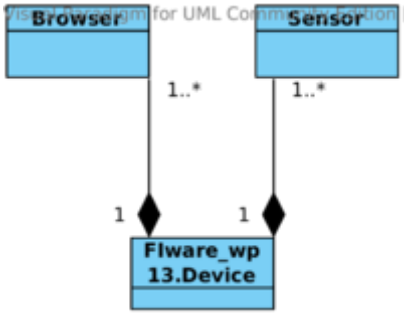
```
    http://www.example.com:PORT/getLocationImageData GET
```

- Start and shutdown the websocket server

```
    http://www.example.com:PORT/closewebsocketserver GET
    http://www.example.com:PORT/postImage            POST
```

This section is for developers who wish to extend the functionality of the JavaScript API and for those who expect to deploy and extend the python Rest server. First part of this section briefly explain the Javascript class structure and how to add a new sensor to it. Second part describe the Python server and its functionality.

15.3.3 Customizing Javascript API



As the image depict Flware_wp13.Device is the main focus of the document and the API is developed around this object. On creation of a Device object a browser is created for each which reads information from the Device API of the browser and make the information available for the Device object. Current API supports GPS, Accelerometer compass and camera.

- Browser object structure is as follows.

```
   var Browser = function(){
   };
   Browser.prototype = {
        isWebSocketSupported : function() {},
        getBrowserType : function() {},
        isDeviceOrientationSupported : function() {},
        isDeviceMotionSupported : function (){},
        isGeolocationSupported : function(){},
        hasVideoCameraSupport : function() {},
        supportedMediaList : function(){
            return { "GPS" :GPS , "Compass" : compass ,
"Accelerometer" : accelerometer, "Video" : camera };
        },
   };
```

Adding a new sensor requires adding a new function isXXXXXSupported() to the browser class and include this sensor in the supportedMediaList() function. This is the **Step 1 of adding a new sensor**

- For each sensor supported there is a Sensor object created and the Structure of the sensor object is as follows.

```
   var Sensor = function(name){
        this.name = name;
        this.array =new Array();
        this.currentValue=0;
   };
   Sensor.prototype = {
        getName : function(){},
        setCurrentValue : function(value){},
```

```
        getCurrentValue : function(){},
        addValue : function(value){
            if(this.name ==="GPS") {
                 this.array.push(value);
            } else if(this.name ==="Compass"){
                 var alpha;
                 var beta;
                 var gamma;
            } else if(this.name ==="Accelerometer"){
            For accelerometer you need 3 arrays to be managed.
             Acceleration this.array[0]
             Acceleration with gravity this.array[1]
             Rotation this.array[2]
            }
         getAdjustedValue :  function(){},
         getValues : function(){},
    };
```

For each Sensor available for users to manipulate there is a sensor object created and object is arranged in this way that all the sensors can be accessed similarly. In achieving this goal the addValue method needs to be set accordingly. This depends on how many data sets and how many values of each data set needs to be stored for the use of the application. For example an application that tracks once movements in a 3D context will need to store all the GPS locations in an array as long as the application. Local database can be used for this purpose in long run but for the performance reasons it is wise to keep certain amount of data in the memory. Another example is compass which needs three sets of data to be stored to follow movements around x, y and z axis respectively. Setting the add value function is **essential stpe 2** of extending the js_api library.

- For each Sensor we depend on the browser Device API to obtain values. **Third step is to add necessary functions** to obtain values from the browser. For example for GPS sensor this is done as follows. This is done in the device object.

```
    FIware_wp13.Device = function(localurl, resturl, localport, wsp
,restport ) {};
    FIware_wp13.Device.prototype = {
        registerForDeviceMovements :
function(onLocationSuccess,onLocationError,options) {}
        getCurrentLocation : function (callback,options){}
    }
```

These functions act as the API functions for the user.If Browsers Media list is updated accurately the necessary data arrays are created automatically and developer needs to update device's setValues() function as follows.

```
    var list = this.getSensorList();
    var count;
```

```
for(count = 0 ; count < list.length; count++ ){
   var sn = list[count].getName();
   if(sn ==="GPS"){
   }else if(sn ==="Accelerometer"){}
   else if(sn ==="Compass"){}
   else {}
}
```

Advantage here in following this method is that sensor data can be manipulated to be more informative with in this function. For example set of GPS locations can be used to extract actual movement directions and velocity information. This also assists in making API browser independent by handling each case with in the API.

- **Forth step is to add the sensor to the metadata tag**. In Current implementation the meta information tag looks like as follows.

```
var metadata={
    type:"image",
    time:time,
    ext:"png",
    devicetype : this.OS,
    browsertype : this.browser.getBrowserType(),
    position : {
        lon:currentGPSValue.longitude,
        lat:currentGPSValue.latitude,
        alt:currentGPSValue.altitude,
        cc:currentGPSValue.accuracy,
   },
   device : {
       ax:currentAcceleration.x,
       ay:currentAcceleration.y,
       az:currentAcceleration.z,
       gx:currentAccelerationWithGravity.x,
       gy:currentAccelerationWithGravity.y,
       gz:currentAccelerationWithGravity.z
       ra:alpha,
       rb:beta,
       rg:gamma,
       orientation:mode,
   },
       vwidth:this.videoelement.videoWidth,
       vheight:this.videoelement.videoHeight,
```

```
        };
```

If this information is related to the 3D context Back end database needs to be updated accordingly. How to do this is described in the administration and installation Guides of this server.

15.3.4 RESTfull server

This server is implemented as a Flask application so developers guide to is just as simple as "Implement necessary flask methods." Functionality of this server is used by the "total mobile client" and the server can be administered using the "REST Pub sub Client and server manager" web page. Main functionality which is uploading and image tagging service can be accessed at once the python script is run.

```
    POST www.example.org:Rest_Port/postBinaryImage
```

In using the above mentioned url to post images server expects the messsage to be organized to as follows

---

| header_length 3 digits | Header-Metadata | Imagedata |

Header length is a 3 digit number that indicates the length of the json message used as metadata. This indicates the length of the json message that is attached as the Header Metadata. Json message is of the format described in the previous section and includes meta data on the image.Image data is the screen capture obtianed from the video element.

In order to extend the functionality methods can be added according to the flask user guide. Flask API docs can be found here

16 Augmented Reality - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

16.1 Introduction

This document introduces how to use the augmented reality JavaScript API. It gives a description of the API functionality and how it is used to implement augmented reality web applications.

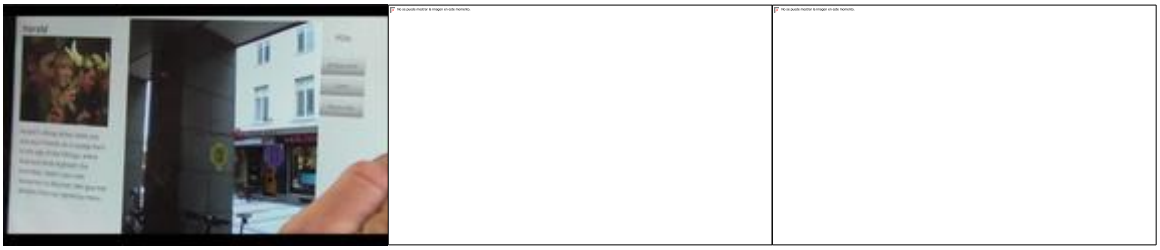
16.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Augmented Reality GE which is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

16.2 User guide

There is no general user interface, because the content and functionality of an AR application varies by implementation, and so does the user interface. User can access AR applications using a web browser, that supports the required features.

Here is some example UIs from the demos.



AR_POI house house

16.3 Programmers guide

This is a guide how to implement Augmented Reality applications using the provided javaScript APIs: Sensor, AR, Scene and Communication, see Augmented Reality Open API Specification.

The API architecture is modular and each API is independent. Therefore, one can use only the APIs needed in a specific application. For example vision based marker tracking application would require AR and Scene APIs, and a location based application, that fetches information from POI Data Provider would require Sensor, Scene, and Communication APIs. Example applications can be found at demos. The APIs are developed and tested on Firefox Nightly, both on mobile devices, and desktop computers. Hence it's **highly recommended** to use **Mozilla Firefox 25.0a1 or higher**. The Sensor API may fail to get requested sensor values, usually on desktop and most of the laptop computers as they do not comprise of the necessary sensors, although the browser might support the events.

In order to use an API, it's set up in the following way:

```
AR.start();
//Create only the APIs needed in your application.
var sensorManager = AR.setupSensors();
var communication = AR.setupConnection();
var ARManager = AR.setupARManager();
```

```
var sceneManager = AR.setupSceneManager();
```

If one would use the AR API, make sure to include the following piece of code inside the xml3d tag into the HTML document.

```
 <data id="MarkerDetector"
     compute="Marker5x5Transforms, Marker3x3Transforms,
imageMarkerTransforms,
     Marker5x5Visibilities, Marker3x3Visibilities,
imageMarkerVisibilities, perspective
     = detect(arvideo, Marker5x5, Marker3x3, imageMarkers,
allowedImageMarkerErrors, flip)">
          <bool name="flip">false</bool>
          <int name="allowedImageMarkerErrors">150</int>
          <int name="imageMarkers">1 -1</int>
          <int name="Marker3x3">32 24 -1 -1 -1 -1</int>
          <int name="Marker5x5">1</int>
          <texture name="arvideo">
              <video autoplay="false"></video>
          </texture>
 </data>
```

Marker3x3, Marker5x5, imageMarkers parameters are used to initialize the IDs that Alvar tracks from the start, they also defines the maximum size of the marker set. The above example tracks the marker3x3 ids 32 and 24 and the maximum set of marker3x3 set is six. Four more marker ids can be later added dynamically using the addMarker function, with marker type 'Marker3x3'.

If one wolud use imageMarkers, make sure to also include the following piece of code into the HTML document.

```
 <div id="imageContainer" style="display: none;">
      <div>../../../imageMarkers/bottle.bmp</div>
      <div>http://chiru.cie.fi/FI-WARE/cebit/assets/120.png</div>
  </div>
```

It defines the urls for the image markers. The value of the imageMarkers parameter is an index to urls in imageContainer. In the above example, Alvar tracks the 120.png image from start.

The perspective value contains a perspective projection matrix created by Alvar and should be passed to view element in HTML document.

```
<view id="View" perspective="#MarkerDetector" />
```

All input parameters for the detect Xflow operator are optional meaning that the parameters' XHTML elements must exist but the element values can be empty.

```
 arvideo:                  The type of this parameter is XML3D texture
and in this case should contain video stream from devices local camera.


 Marker5x5:                The type of this parameter is XML3D int and
it must contain zero or more IDs of ALVAR JavaScript's built-in
```

5x5Markers in a whitespace-separated list. These markers can be created with a separate marker creator which can be

downloaded from [1] as a part of the ALVAR library.

Marker3x3: The type of this parameter is XML3D int and it must contain zero or more IDs of ALVAR JavaScript's built-in

3x3Markers in a whitespace-separated list. The IDs of these markers are between values of 0 and 63.

imageMarkers: The type of this parameter is XML3D int and it must contain zero or more custom image marker IDs in a

whitespace-separated list. The IDs can be same as ALVAR JavaScript's built-in marker IDs.

The only limitation for the image markers is that they need to be square and contain a black border.

Testing the suitable border width and image marker content is up to the application developer,

since these properties depend on the nature of the application and the usage environment.

The marker image is converted to grayscale format in ALVAR JavaScript so the color information is discarded.

allowedImageMarkerErrors: The type of this parameter is XML3D int and it must contain zero or more whitespace-separated values of allowed

errors to be used in image marker detection. Each value is related to a value with the same position in the

imageMarkers list. For example, the second value in the allowedImageMarkers is the error value for the

second marker ID in the imageMarkers. Thus, the marker IDs, to which the application developer wants to define

a custom error value, should be put in the beginning of the imageMarkers list. The error values are relative to the

used image marker content complexity so the application developer should try out which error

values suit best for different markers. The default value for an image marker error in ALVAR JavaScript is 0.625

times marker's width/height. This is used if all or some of the values are left empty.

flip: The type of this parameter is XML3D bool and its value can be either true or false, the default value being false.

```
                              When the value is true, the resulting
transform matrices from marker detection are flipped so that they
respond to
                              flipped camera feed.
```

Example of different types of allowed markers:

Marker3x3 Marker5x5 ImageMarker ImageMarker ImageMarker

16.3.1 Sensor API

Sensor API is used for creating sensor listeners. The Sensor API is based on the following W3C specifications Geolocation, DeviceOrientation, DeviceLight, DeviceProximity.

The supported sensor types:

```
SensorType
  orientation
  motion
  light
  proximity
```

- getAvailableSensors()

  Returns an array of available sensor types.

- getSensorListeners()

  Returns a dictionary of currently active sensor listeners.

- listenSensor(sensorType)

  Returns sensor listener for the given sensor type.

  For example: listen device orientation and use it to rotate the virtual camera.

```
orientationListener = sensorManager.listenSensor('orientation');
orientationListener.addAction(sceneManager.setCameraOrientation);
```

- hasGPS()

  Returns true if the device has a GPS sensor.

- getCurrentPosition(successCallback, errorCallback, options)

  Attaches the given callback functions to "one-shot" position request. Uses the HTML5 Geolocation API getCurrentPosition() method to get the device's position.

  For example: Get POIs nearby. The getPois function is defined at the example for queryData function in Communication API

```
sensorManager.getCurrentPosition(getPOIs);
```

- watchPosition(successCallback, errorCallback, options)

Attaches the given success callback function to updated position as the device moves. Uses the HTML5 Geolocation API getCurrentPosition() method to get the device's position updates.

16.3.2 AR API

AR API is used for registering and tracking markers.

- setMarkerCallback(callback)

  Sets a callback function for detected markers, the function has six input parameters callBackFunction(Marker3x3Transforms, Marker5x5Transforms, imageMarkerTransforms, Marker3x3Visibilities, Marker5x5Visibilities, imageMarkerVisibilities).

- addMarker(markerId, markerType)

  Adds the given marker id, into the set of markers that Alvar tracks.

  markerType can be Marker3x3, Marker5x5, imageMarker.

16.3.3 Scene API

Scene API is used for manipulating the elements in a xml3d scene. The actual xml3d scene can be defined in the web page using tags such as, mesh, group, transform, view, shader, etc. More information about how to use the xml3d can be found here: XML3D Open API Specification

- setPositionFromGeoLocation(curLoc, elemLoc, xml3dElement, minDistance, maxDistance)

  Positions the given xml3dElement(virtual object) into the virtual scene by using the given parameters: curLoc is the current gps location of the device, elemLoc is the gps location of the xml3dElement, and the calculated distance is clamped between minDistance and maxDistance.

- setCameraOrientation(deviceOrientation)

  Replaces the existing orientation of the virtual camera with the given device orientation.

- translateCameraFromGps(curLoc, gpsPoint, maxStep)

  Translates the camera from current gps location to gpsPoint. The translation is discarded if the distance between the current and new location exceeds the maxStep.

- translateCameraFromMotion(deviceMotion)

  Translates the camera according to the acceleration from deviceMotion event.

- setCameraMotionTranslationStepSize(stepSize)

  The given step size value defines the resolution of the virtual camera movement.

  The default value is **1.0**.

- setCameraDegreesOfFreedom(heave, sway, surge, yaw, pitch, roll)

  The given Boolean parameters define the freedom of degrees that the virtual camera currently has.

  heave: allows virtual camera to move up and down. The default value is **false**.
  sway: allows virtual camera to move left and right. The default value is **false**
  surge: allows virtual camera to move forward and backward. The default value is **true**.
  yaw: allows virtual camera to rotate around y-axis. The default value is **true**.
  pitch: allows virtual camera to rotate around x-axis. The default value is **true**.
  roll: allows virtual camera to rotate around z-axis. The default value is **false**.

- setTransformFromMarker(markerTransform, xml3dElement, rotateX)

Sets the given marker transform to the given xml3dElement. if rotateX is true the given xml3dElement is rotated 90 degrees.

- setCameraVerticalPlane(degrees)

  Sets the camera vertical plane into the given input degrees.

- addObjectToBillboardSet(xml3dElement)

  Adds the given xml3dElement into a billboard set. Objects that belong to billboard set, are always facing towards the virtual camera.

- getActiveCamera()

  Returns the xml3d active view element.

- getDistance(gpsPoint1, gpsPoint2)

  Returns the distance(meters) and bearing(radians) between the given gps coordinates.

16.3.4 Communication API

Communication API is used for handling the basic communication with remote services(Other GEs).

- addRemoteService(serviceName, sourceURL)

  Adds a new remote service with the given service name and url. Remote service, such as POI Data Provider, must provide a RESTful API for communication.

  For example: Add a POI Data Provider.

```
communication.addRemoteService("POI_Data_Provider", "http://someUrl");
```

- queryData(serviceName, restOptions, successCallback, errorCallback)

  Builds the REST query based on the given REST options and sends XMLHttpRequest to the given remote service. If the query is successful, the success callback function will handle the remote service's response message.

  For example: Query data from the POI Data Provider, added earlier.

```
function getPOIs(gpsCoordinates)
    var restOptions = {
        'function' : "radial_search",
        'lat' : gpsCoordinates.latitude,
        'lon' : gpsCoordinates.longitude,
        'category' : "cafe",
        'radius' : 1500
    }
    communication.queryData("POI_Data_Provider", restOptions,
handlePoi, null);
}
```

- sendData(serviceName, message, succesCallback, errorCallback)

  Sends the given message to the given remote service.

- listenWebsocket(url)

  Opens a websocket and connects it to given url.

17 Real Virtual Interaction - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

17.1 Introduction

This documents describes instructions on how to use demo applications designed for Real Virtual Interaction backend, which is the primary deliverable of FIWARE.OpenSpecification.MiWi.RealVirtualInteraction GE. For server installation, please refer to Real Virtual Interaction - Installation and Administration Guide. This guide consist of an Android application for simulating actuators and sensors, and publish/subscribe and request/response clients. The full architecture and deployment diagram can be seen in Figure 1 in FIWARE.OpenSpecification.MiWi.RealVirtualInteraction.

This document is divided in to three parts:

- Real Virtual Sensor Simulator (Android application)
- Publish Subscribe client (JavaScript web application)
- Request Response client (HTML web application)

Source codes (for latest version of code):

- RealVirtual-AndroidApp
- RealVirtual-WebClients

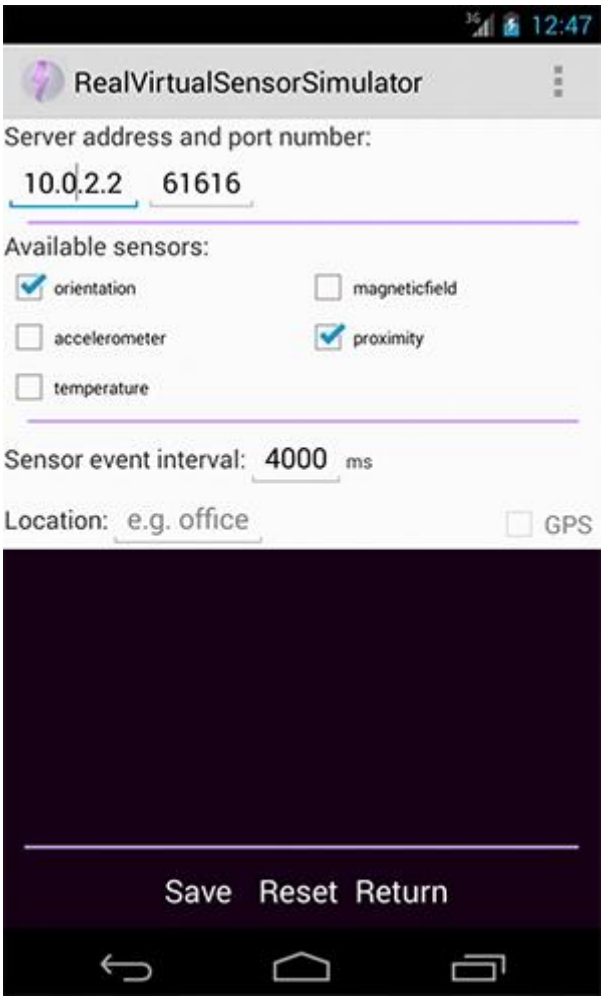
This User and Programmers Guide relates to the XYZ GE which is part of the Advanced Middleware and Web User Interfaces chapter. For more background information on this GE, also refer to its Open Specification and Architecture Description.

17.2 User guide

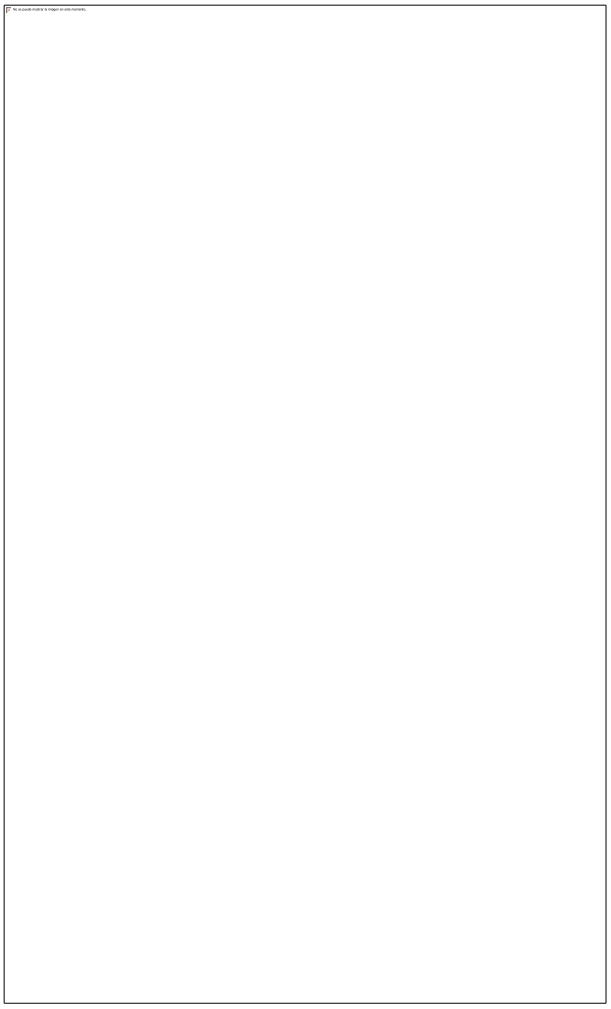
The user guides are meant for end-users using these application to test the Real Virtual Interaction Server. This guide includes information regarding usage of these applications.

17.2.1 Real Virtual Sensor Simulator

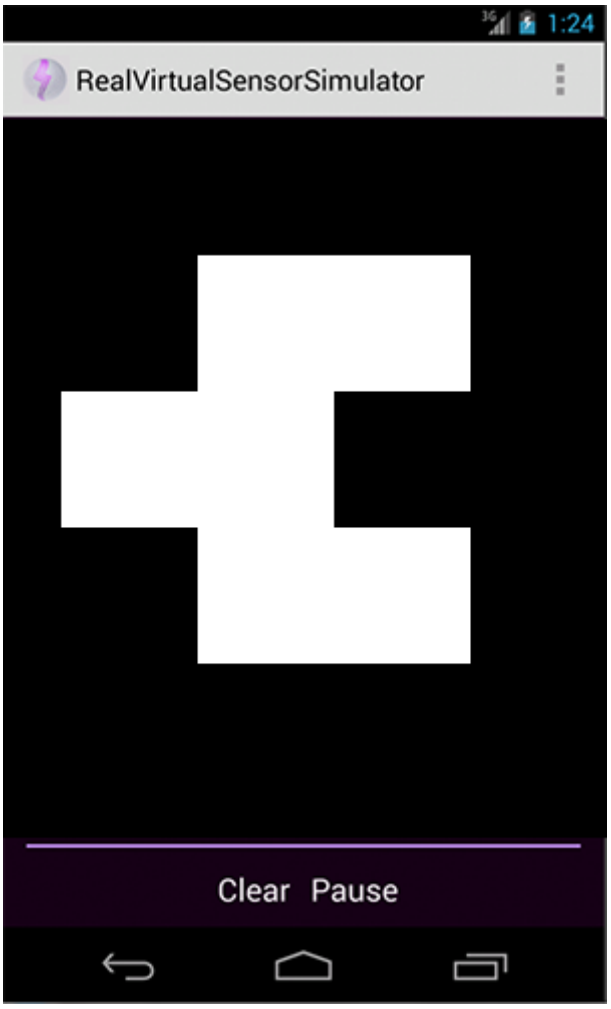
This Android application is designed to be a demo application for the Real Virtual Interaction backend. The application reads sensors available on an android device and sends this information to the Real Virtual Interaction backend which stores this information to be used by other demo applications such as the publish-subscribe client (JavaScript web application) and the request-response client (HTML web application). In essence this application emulates real world sensor data. Also this application offers one "actuator" which is the display. This Android application requires API level 14 device to work (see further details about suitable devices here). Also user needs to enable network access and preferably GPS (optional).



From settings view end-user can configure what sensors are to be listened and send to real virtual interaction server. Also user can define a custom connection information. Available sensors are detected from the Android device and updated dynamically on screen. The view supports changing orientation and uses shared preferences to save user data. The location field can be used to give textual representation in where this particular device is currently. It can be a name of a room, building or address. It is up to client developers to make use of this data. When user clicks the GPS check box system will try to automatically enable GPS, but this is not supported in all Android versions. The GPS check box will be disable if there is no GPS available in the system.



The main view shows the communication between the client and the server. In main view the text colored as "cyan" show outbound messages. Due to nature of UDP traffic all sent messages are being shown, but there is no verification whether these messages have reached the server. Received messages are being shown in "yellow" color. The idea is that the device will publish sensor and actuator interfaces using the RESTful data format, as described in FIWARE.OpenSpecification.MiWi.RealVirtualInteraction.



This view represent a case where device display acts as an actuator where a remote HTTP POST call can be used to trigger the marker view display and what marker is to be shown. The application contains 60 ready made augmented reality markers used by the software in FIWARE.OpenSpecification.MiWi.POIDataProvider. HTTP POST calls are being send to the real virtual interaction backend which will extract the query part and pass the message to last known IP address of the connected Android application. Android application will parse the received message and if it is in the right format, it will automatically change main view to show the marker requested in the query string. Otherwise the message will be display as a general inbound message with "yellow" color on the main view. Sample query practise is being shown in FIWARE.OpenSpecification.Details.MiWi.RealVirtualInteraction in more detail.

17.2.2 Publish/Subscribe Web Application

This application is designed to receive and send data to the Android application by using the Real Virtual Interaction backend as a mediator.

The view is divided in two sections. The section in right appends information received from each individual sensor as Jquery tab widgets. By clicking the header of tab you can expand the tab to show information within that tab and thus from individual sensor. There are some basic data implemented in the current design. When new event is being received from the server these tabs get redrawn. If new sensors associated with selected device arrive new tabs are being appended to this view. During this update all new values are being appended to existing tabs as well. Each tab holds history for 5 latest events. Android Developers offers detailed description about sensors events. The left view show the visualization of the events. The application currently supports three formats "double, array, 3DPoint". These are being referred as primitives in the code. The decision about which type of graph to draw depends on the primitive type of the data. This information is being embedded in the RESTful data format.

The title of tab acts as a link that opens a visual representation of the data being withhold in the particular tab/sensor. Keep in mind that the link is a standard href and is only triggered from pixels of link text and thus may require few clicks occasionally.

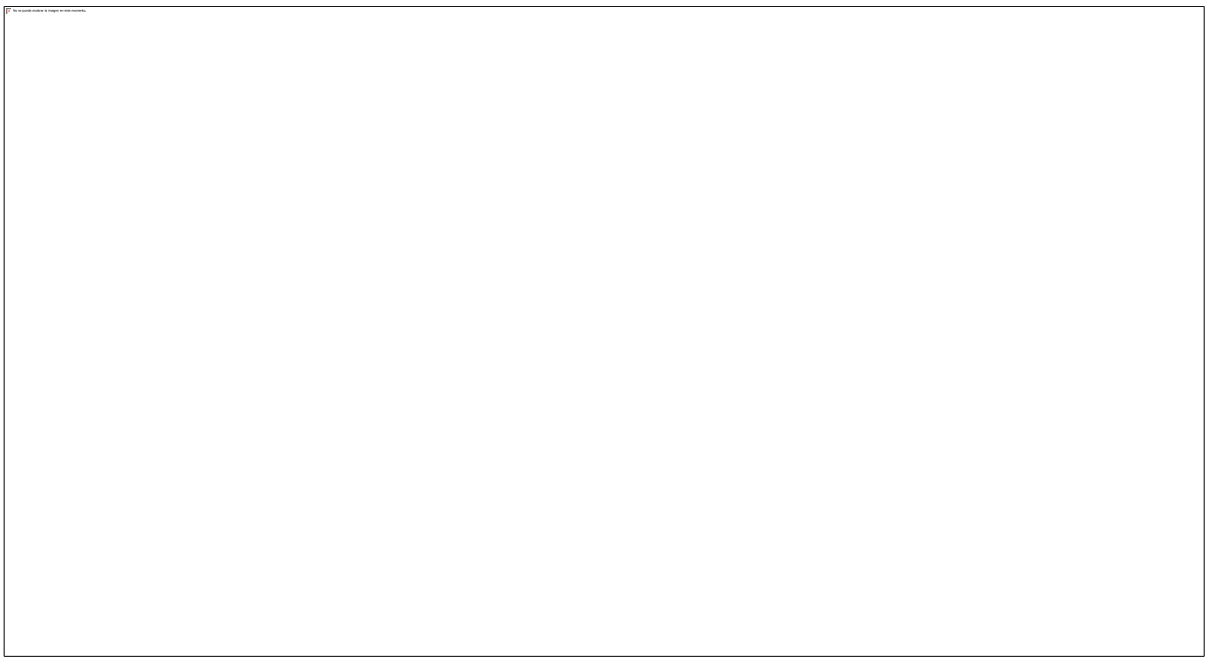
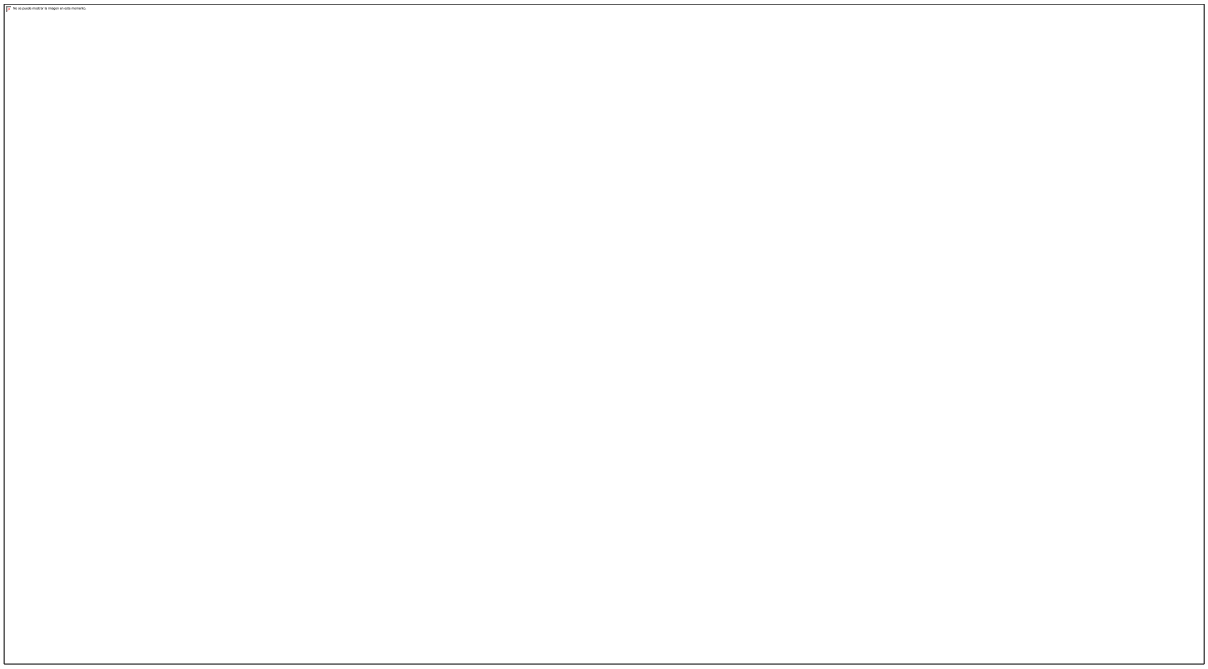


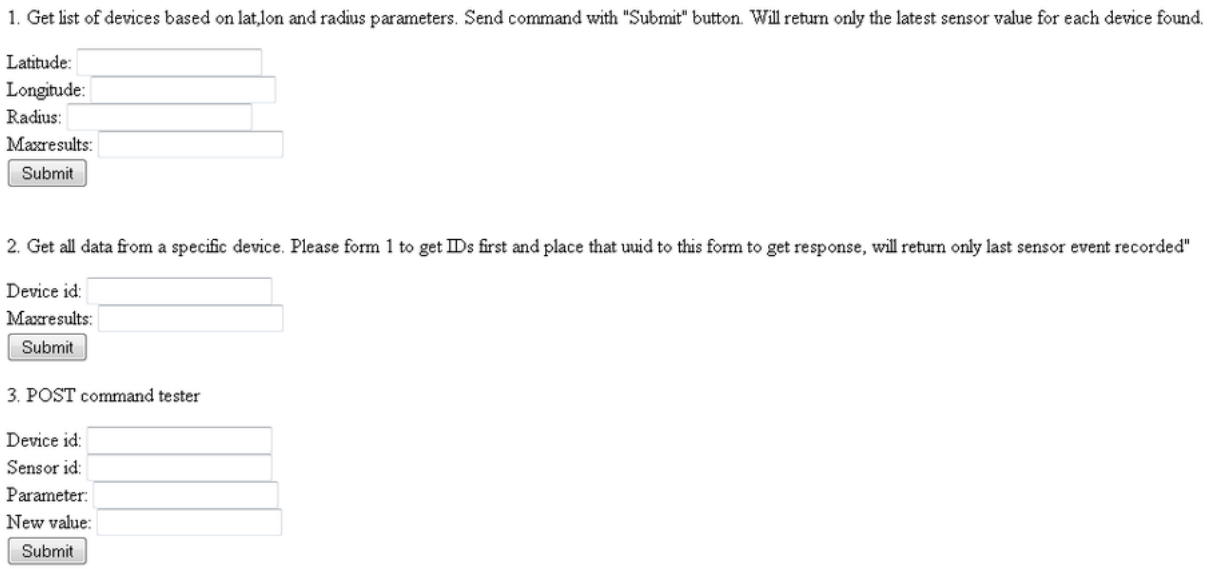
Figure 4 shows a view for all single entry values such represented by "primitive" : "double" in the JSON string of that particular sensor. 5 latest values are being stored in the view each represented by a 3D bar. When 6th event arrive the oldest entry is removed and new added right of the coordinate system and each bar moved to left accordingly. This update is being seen live if the view is drawn while new event arrives to that particular sensor. Keep in mind that that the view is redrawn even when there is an event coming for ANY sensor.

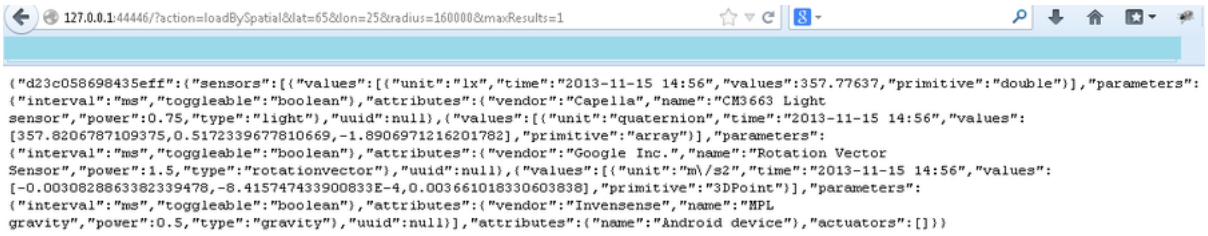


The view in Figure 5 shows a standard representation style for all "3Dpoint" type of arrays where each element represents X,Y,Z axis for instance.

17.2.3 Request/Response Web Application

This application is a sample page which holds standard html form elements. One purpose of this page is to enable user to demo that the server actually is receiving the data from the Android application. Also this page acts as a sample of a 3rd party service querying data from the Real Virtual Interaction backend. Using the RESTful data format it could be possible to dynamically embed new html form elements in to a web page. For instance, the case 3 showing the POST call form element could be loaded with information ready on the input fields when the web page is being loaded by the user. More detailed description on what is happening on the backend can be found from FIWARE.OpenSpecification.Details.MiWi.RealVirtualInteraction.





JSONLint is a good tool for validating and organizing the response query in more easily readable form.

17.2.4 Known Issues

If the sensor event arrive to PubSub client faster than it is able finish a render cycle there will be problems with drawing and might cause instability depending on browser. Some browser might have a fixed FPS limit and thus the minimum interval sensor events should be 1000 ms / MAX_FRAME_RATE. The current application does not have any logic to handle such occurance.

Not all browsers support Web Socket or Web GL. Current versions of Chrome and FireFox have been tested to be working.

The PubSub web application and Android application might not handle small screens very well. With small screen it is possible that the 3D visualization of web application might be drawn in wrong location and in Android some screen sizes have not been tested.

17.3 Programmers guide

The Real Virtual Sensor Simulator Android software can be imported in to an Eclipse IDE as an Android project. You need to also download Android SDK and ADT plugin. Only way internal configurations can be changes is to download the source code and change them and compile a new .apk file. For using RealVirtuaInteraction backend, please have a look at the API specifications FIWARE.OpenSpecification.Details.MiWi.RealVirtualInteraction .

You can easily change the IP address if you wish to test the publish/subscrice client with the server locally or deploy it remotely by changing the IP address.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. The key word REMOTEHOST can be replaced by either IP address or DNS host name. Also port number for middleware service can differ from IANA standard which is 80 for the WebSocket and 8080 for HTTP. The key words GET,POST,PUT and DELETE are http methods and appear capitalized each time they occur in the specifications.

Security implementations are not included in this specifications as they are highly dependable on type of middleware service and chosen security level. For controlled public access api-keys or session-ids could be used. Alternatively for private access login information could be included in queries.

API examples for device application developers

The Realvirtualinteraction backend will listen to incoming UDP packets and will drop packets that do not conform with the RESTful data format specification (version 1.0). The payload string MAY be Gzip compressed. Below JSON string is a example how the device developers should public sensor/actuator information to the server. The "dataformat_version" field will be removed after the packet is being received by the server and will not be passed on to possible clients subscribed listening for incoming events. For instance the existing logic could be extended to include other fields such as API-KEY to ensure that only registered devices may publish to server. This could be the first step to add a layer of security.

```
{
    "dataformat_version": "1.0",
    "d23c058698435eff": {
        "d23c058698435eff": {
            "sensors": [
                {
                    "value": {
                        "unit": "uT",
                        "primitive": "3DPoint",
                        "time": "2014-02-19 09:40:06",
                        "values": [
                            17.819183349609375,
                            0.07265311479568481,
                            -0.4838427007198334
                        ]
```

```
                    },
                    "configuration": [
                        {
                            "interval": "ms",
                            "toggleable": "boolean"
                        }
                    ],
                    "attributes": {
                        "type": "orientation",
                        "power": 0.5,
                        "vendor": "Invensense",
                        "name": "MPL magnetic field"
                    }
                },
                {
                    "value": {
                        "unit": "uT",
                        "primitive": "3DPoint",
                        "time": "2014-02-19 09:40:06",
                        "values": [
                            17.819183349609375,
                            0.07265311479568481,
                            -0.4838427007198334
                        ]
                    },
                    "configuration": [
                        {
                            "interval": "ms",
                            "toggleable": "boolean"
                        }
                    ],
                    "attributes": {
                        "type": "gyroscope",
                        "power": 0.5,
                        "vendor": "Invensense",
                        "name": "MPL magnetic field"
                    }
                },
```

```
                {
                    "value": {
                        "unit": "uT",
                        "primitive": "3DPoint",
                        "time": "2014-02-19 09:40:06",
                        "values": [
                            17.819183349609375,
                            0.07265311479568481,
                            -0.4838427007198334
                        ]
                    },
                    "configuration": [
                        {
                            "interval": "ms",
                            "toggleable": "boolean"
                        }
                    ],
                    "attributes": {
                        "type": "magneticfield",
                        "power": 0.5,
                        "vendor": "Invensense",
                        "name": "MPL magnetic field"
                    }
                },
                {
                    "value": {
                        "unit": "m/s2",
                        "primitive": "3DPoint",
                        "time": "2014-02-19 09:40:06",
                        "values": [
                            0.006436614785343409,
                            0.003891906701028347,
                            -0.5983058214187622
                        ]
                    },
                    "configuration": [
                        {
                            "interval": "ms",
```

```
                            "toggleable": "boolean"
                        }
                    ],
                    "attributes": {
                        "type": "linearacceleration",
                        "power": 1.5,
                        "vendor": "Google Inc.",
                        "name": "Linear Acceleration Sensor"
                    }
                }
            ],
            "actuators": [
                {
                    "configuration": [
                        {
                            "value": "100",
                            "unit": "percent",
                            "name": "viewsize"
                        }
                    ],
                    "actions": [
                        {
                            "value":
"[marker1,marker2,marker3,marker4,marker6,marker7,marker8,marker9,marke
r10,marker11,marker12,marker13,marker14,marker15,marker15,marker16,mark
er17,marker18,marker19]",
                            "primitive": "array",
                            "unit": "string",
                            "parameter": "viewstate"
                        }
                    ],
                    "callbacks": [
                        {
                            "target": "viewstate",
                            "return_type": "boolean"
                        }
                    ],
                    "attributes": {
                        "dimensions": "[480,800]"
```

```
                    }
                }
            ],
            "attributes": {
                "name": "Android device"
            }
        }
    }
}
```

API examples for web client application developers

Following code and header samples enable a real-time connection over TCP/IP to be formed with a server application. Once a connection is established, sensor events MAY be pushed to clients from server in real-time. The connection is full-duplex meaning that also a client MAY send messages directly to sensors in through a web server. This sort of full-duplex connection MAY be considered as publish/subscribe type of connection where client MAY choose which sensor to subscribe to receive event updates from. The web service SHALL provide the client a list of available sensors or OPTIONALLY a client MAY use third party service such as point-of-interest(POI) service to find sensors. WebSocket SHOULD be then used to form direct connection to the sensors through a IoT.Broker type of web server component.

JavaScript client sample:

```
function CreateWebSocket() {

      if ("WebSocket" in window) {
         ws = new WebSocket("ws://REMOTEHOST");

         ws.onopen = function() {
           alert("Connection established to web server");
         };
         ws.onmessage = function (evt) {
           alert("Message received from web server: " + evt.data);
         };
         ws.onclose = function() {
            alert("Connection is closed...");
         };
      }
      else {
         alert("WebSocket NOT supported by your Browser!");
      }
}
```

The above JavaScript example initiates a connection to a webserver and starts handshake with following HTTP header.

REQUEST HEADER:

```
GET /chat HTTP/1.1
Host: REMOTEHOST
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: LOCALHOST
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Server MUST respond with following HTTP header or handshake fails. Notice that the Sec-WebSocket-Accept key is unique and MUST be created by server instance. Detailed instructions can be found from RFC6455.

RESPONSE HEADER:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

API examples for service developpers:

Following example show how backend services SHALL communicate between each other using HTTP GET/POST methods. OPTIONALLY other HTTP methods such as PUT and DELETE MAY be used, but they are not supported by the real virtual interaction backend deliverable.

1. Request all sensors with bound by a specific spatial bounds

A middleware web service SHOULD offer ways for other middleware services to specify retrievable devices by location and spatial bounds or OPTIONALLY by an IP address space. The spatial bound SHALL be either a square area with minimum and maximum values for coordinates, a circle with a centerpoint and radius or a complex shape.

Following example shows an example where POI middleware service requests all devices available within a specific circular area with a geo-coordinate center point and radius in meters.

Below is a sample code that can be used to form the following request query:

```
<form action="http://127.0.0.1:44446/" method="get">
<input type="hidden" name="action" value="loadById">
Device id: <input type="text" name="device_id"><br>
Maxresults: <input type="text" name="maxResults"><br>
```

```
<input type="submit" value="Submit">
</form>
```

Below is a sample request header as received by the real virtual interaction backend:

REQUEST HEADER:

```
GET /?action=loadBySpatial&lat=65.4&lon=25.4&radius=1500&maxResults=1
HTTP/1.1
Host: 127.0.0.1:44446
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0)
Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Below is a sample reponse header as send by the real virtual interaction backend.

RESPONSE HEADER:

```
 HTTP/1.1 200 OK
 Access-Control-Allow-Origin: *
 Content-Type: text/plain; charset=utf-8
 Content-Length: 1767

 Connection: close

{
    "d23c058698435eff": {
        "sensors": [
            {
                "value": {
                    "unit": "uT",
                    "primitive": "3DPoint",
                    "time": "2013-12-10 15:02:39",
                    "values": [
                        0.07543107122182846,
                        -0.015922529622912407,
                        0.01725415326654911
                    ]
                },
                "configuration": [
```

```
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ],
                "attributes": {
                    "type": "orientation",
                    "power": 0.5,
                    "vendor": "Invensense",
                    "name": "MPL magnetic field"
                }
            },
            {
                "value": {
                    "unit": "rad/s",
                    "primitive": "3DPoint",
                    "time": "2013-12-10 15:02:39",
                    "values": [
                        355.9173278808594,
                        -85.8130111694336,
                        4.165353775024414
                    ]
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ],
                "attributes": {
                    "type": "gyroscope",
                    "power": 0.5,
                    "vendor": "Invensense",
                    "name": "MPL Gyro"
                }
            },
            {
                "value": {
```

```
                    "unit": "uT",
                    "primitive": "3DPoint",
                    "time": "2013-12-10 15:02:39",
                    "values": [
                        0.07543107122182846,
                        -0.015922529622912407,
                        0.01725415326654911
                    ]
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
                ],
                "attributes": {
                    "type": "magneticfield",
                    "power": 0.5,
                    "vendor": "Invensense",
                    "name": "MPL magnetic field"
                }
            },
            {
                "value": {
                    "unit": "m/s2",
                    "primitive": "3DPoint",
                    "time": "2013-12-10 15:02:39",
                    "values": [
                        352.0169982910156,
                        -85.75300598144531,
                        4.191023826599121
                    ]
                },
                "configuration": [
                    {
                        "interval": "ms",
                        "toggleable": "boolean"
                    }
```

```
                ],
                "attributes": {
                    "type": "linearacceleration",
                    "power": 1.5,
                    "vendor": "Google Inc.",
                    "name": "Linear Acceleration Sensor"
                }
            }
        ],
        "actuators": [
            {
                "configuration": [
                    {
                        "value": "100",
                        "unit": "percent",
                        "name": "viewsize"
                    }
                ],
                "actions": [
                    {
                        "value":
"[marker1,marker2,marker3,marker4,marker6,marker7,marker8,marker9,marke
r10,marker11,marker12,marker13,marker14,marker15,marker15,marker16,mark
er17,marker18,marker19]",
                        "primitive": "array",
                        "unit": "string",
                        "parameter": "viewstate"
                    }
                ],
                "callbacks": [
                    {
                        "target": "viewstate",
                        "return_type": "boolean"
                    }
                ],
                "attributes": {
                    "dimensions": "[480,800]"
                }
            }
```

```
        ],
        "attributes": {
            "name": "Android device"
        }
    }
}
```

The response header returns radius and geo-coordinates which were set by the original request query. *Devices* JSONArray object contains all devices matching the query. Each device MAY contain multiple sensors and actuators.

2. Request all data from a specific device by device UUID

A device SHOULD be considered as a micro controller board with capabilitis required to generate an *uuid*. A device MAY contain any number of sensors and actuators, and in any combination. If the requested sensor or actuator does not have uuid the request MUST target the device containing the desired sensor or actuator.

Following example shows how a middleware service retrieves all available information regarding a specific device by using an *uuid* string identifier.

Below is a sample code that can be used to form the following request query:

```
<form action="http://127.0.0.1:44446/" method="get">
<input type="hidden" name="action" value="loadBySpatial">
Latitude: <input type="text" name="lat"><br>
Longitude: <input type="text" name="lon"><br>
Radius: <input type="text" name="radius"><br>
Maxresults: <input type="text" name="maxResults"><br>
<input type="submit" value="Submit">
</form>

Below is a sample request header as received by the real virtual
interaction backend.
```

REQUEST HEADER:

```
GET /?action=loadById&device_id=440cd2d8c18d7d3a&maxResults=1 HTTP/1.1
Host: 127.0.0.1:44446
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0)
Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

```
Connection: keep-alive
```

Below is a sample response header as send by the real virtual interaction backend.

RESPONSE HEADER:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: text/plain; charset=utf-8
Content-Length: 1767

Connection: close

{
   "440cd2d8c18d7d3a": {
       "actuators": [
           {
               "configuration": [
                   {
                       "unit": "percent",
                       "name": "viewsize",
                       "value": "100"
                   }
               ],
               "callbacks": [
                   {
                       "return_type": "boolean",
                       "target": "viewstate"
                   }
               ],
               "attributes": {
                   "dimensions": "[480,800]"
               },
               "actions": [
                   {
                       "unit": "string",
                       "primitive": "array",
                       "parameter": "viewstate",
                       "value":
"[marker1,marker2,marker3,marker4,marker6,marker7,marker8,marker9,marke
```

```
r10,marker11,marker12,marker13,marker14,marker15,marker15,marker16,mark
er17,marker18,marker19]"
                    }
                ]
            }
        ],
        "sensors": [
            {
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
                ],
                "values": [
                    {
                        "unit": "rads",
                        "primitive": "3DPoint",
                        "values": [
                            21.117462158203125,
                            -0.9801873564720154,
                            -0.6045787930488586
                        ],
                        "time": "2013-12-10 10:07:30"
                    }
                ],
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL Gyro",
                    "power": 0.5,
                    "type": "gyroscope"
                }
            },
            {
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
```

```
                ],
                "values": [
                    {
                        "unit": "ms2",
                        "primitive": "3DPoint",
                        "values": [
                            149.10000610351562,
                            420.20001220703125,
                            -1463.9000244140625
                        ],
                        "time": "2013-12-10 10:07:30"
                    }
                ],
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL accel",
                    "power": 0.5,
                    "type": "accelerometer"
                }
            },
            {
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
                ],
                "values": [
                    {
                        "unit": "uT",
                        "primitive": "3DPoint",
                        "values": [
                            -0.08577163517475128,
                            0.16211289167404175,
                            9.922416687011719
                        ],
                        "time": "2013-12-10 10:07:30"
                    }
```

```
                ],
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL magnetic field",
                    "power": 0.5,
                    "type": "magneticfield"
                }
            },
            {
                "configuration": [
                    {
                        "toggleable": "boolean",
                        "interval": "ms"
                    }
                ],
                "values": [
                    {
                        "unit": "orientation",
                        "primitive": "3DPoint",
                        "values": [
                            -0.004261057823896408,
                            -0.017044231295585632,
                            0.019174760207533836
                        ],
                        "time": "2013-12-10 10:07:30"
                    }
                ],
                "attributes": {
                    "vendor": "Invensense",
                    "name": "MPL Orientation (android deprecated
format)",
                    "power": 9.699999809265137,
                    "type": "orientation"
                }
            }
        ],
        "attributes": {
            "name": "Android device"
        }
```

```
    }
}
```

The above response shows a general description how a JSON object returned by the middleware service could look like.

3. Change value of a specific attribute of a sensor by ID, controller name, new value

Middleware service MUST offer a way to change state of sensors or actuators. Sensors and actuators SHOULD publish configurable parameters. Middleware services MUST send state change requests as HTTP POST calls. POST request content MUST start with action definition.

Following example shows how to use HTTP POST request to turn change augmented reality marker on an Android application remotely.

Below is a sample code that can be used to form the following request query:

```
<form action="http://127.0.0.1:44446/upload" enctype="multipart/form-
data" method="post">
Device id: <input type="text" name="device_id"><br>
Choose marker to upload: <input type="file" name="datafile"
size="40"></br>
<input type="submit" value="Send">
```

REQUEST HEADER:

```
POST / HTTP/1.1
Host: 127.0.0.1:44446
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:25.0)
Gecko/20100101 Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 92

action=update&device_id=440cd2d8c18d7d3a&sensor_id=display&parameter=vi
ewstate&value=marker5
```

RESPONSE HEADER:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: text/plain; charset=utf-8
```

```
Content-Length: 6
Connection: close
200 OK
```

Real virtual interaction backend uses the device_id parameter and looks up the IP address from reference table and passes only the content of the query forward to the particular sensor. If an IP address is found from the reference table, the server will respond with 200 OK without actually knowing whether the message reached its destination as the transport mechanism is UDP. Otherwise server will respond with 404 NOT FOUND.

REQUEST RECEIVED BY SENSOR:

```
action=update&device_id=440cd2d8c18d7d3a&sensor_id=display&parameter=vi
ewstate&value=marker5
```

Possible response codes from sensors

Following list includes those HTTP codes supported by the CoAP protocol. These codes SHOULD be returned by sensors or actuators if they are able. The middleware service MUST respond to all requests even if there is no response from a sensor. In such case the middleware SHALL implement time-out after which a appropriate response MUST be generated.

```
| Code | Description                       | Reference |
+------+-----------------------------------+-----------+
| 2.01 | Created                           | [RFCXXXX] |
| 2.02 | Deleted                           | [RFCXXXX] |
| 2.03 | Valid                             | [RFCXXXX] |
| 2.04 | Changed                           | [RFCXXXX] |
| 2.05 | Content                           | [RFCXXXX] |
| 4.00 | Bad Request                       | [RFCXXXX] |
| 4.01 | Unauthorized                      | [RFCXXXX] |
| 4.02 | Bad Option                        | [RFCXXXX] |
| 4.03 | Forbidden                         | [RFCXXXX] |
| 4.04 | Not Found                         | [RFCXXXX] |
| 4.05 | Method Not Allowed                | [RFCXXXX] |
| 4.06 | Not Acceptable                    | [RFCXXXX] |
| 4.12 | Precondition Failed               | [RFCXXXX] |
| 4.13 | Request Entity Too Large          | [RFCXXXX] |
| 4.15 | Unsupported Content-Format        | [RFCXXXX] |
| 5.00 | Internal Server Error             | [RFCXXXX] |
| 5.01 | Not Implemented                   | [RFCXXXX] |
| 5.02 | Bad Gateway                       | [RFCXXXX] |
| 5.03 | Service Unavailable               | [RFCXXXX] |
| 5.04 | Gateway Timeout                   | [RFCXXXX] |
| 5.05 | Proxying Not Supported            | [RFCXXXX] |
+------+-----------------------------------+-----------+
```

18 Virtual Characters - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

18.1 Introduction

This document describes usage of the Virtual Characters Generic Enabler. It ties with the 3D-UI and Synchronization GE's to implement animating characters, that optionally are synchronized over the network in multi-user networked applications. All three GE's are contained within the WebTundra Javascript codebase.

18.1.1.1 *Background and Detail*

The Virtual Characters GE is part of the Advanced Middleware and Web User Interfaces chapter. Please find more information about this Generic Enabler in the related Open Specification and Architecture Description.

18.2 User guide

This is a developer oriented GE, ie. it allows to include animating 3D characters in web applications. Therefore there is no separate user guide, instead all the functionality is listed under the Programmer's Guide section.

18.3 Programmers guide

The scene model utilized by the WebTundra codebase is component-based. This means that to enable functionality, various *components* are created to scene *entities* which are otherwise just empty containers. The basics of positioning and showing a 3D object happen through the Placeable and Mesh components implemented by the 3D-UI GE.

The animation capabilities are controlled through the AnimationController component. It controls, and automatically updates, the animations of a Mesh component contained in the same entity. This requires the Mesh component to refer to a mesh asset with animation data. The operations available include:

- Animation play / stop
- Speed and direction (forward / backward) control
- Animation fade in / fade out control

There is also a second component, Avatar, which allows automatic instantion of the necessary Placeable, Mesh and AnimationController components by parsing a JSON-format character description file.

18.3.1 JavaScript client library reference

All the client library classes are embedded within the namespace Tundra.

To understand the basics of the rendering and the scene model, also refer to the user guides of the 3D-UI and the Synchronization GE's:

- 3D-UI - WebTundra - User and Programmers Guide
- Synchronization - User and Programmers Guide

18.3.1.1 *AnimationController*

The following functions exist in the AnimationController component for controlling animation playback. Animations are referred to with their string names. The playing animations will be automatically updated in conjunction with rendering each frame.

```
animationController.play(name, fadeInTime, crossFade, looped)
```

Start playback of an animation. fadeInTime specifies a time in seconds during which to smoothly blend the animation from zero blending weight to full weight. crossFade is a boolean which will cause other animations to be faded out as the playback of the new animation is started. looped is a boolean to indicate whether the animation should loop, or only play once.

```
animationController.playLooped(name, fadeInTime, crossFade)
```

Same as above, but the animation will always be looped.

```
animationController.stop(name, fadeOutTime)
```

Stop playback of an animation. fadeOutTime is the fade-out period in seconds, during which the blending weight is smoothly reduced to zero.

```
animationController.stopAll()
```

Stop playback of all animation of the character.

```
animationController.setAnimWeight(name, weight)
```

Set the blending weight of an animation between 0 (none) and 1 (full).

```
animationController.setAnimSpeed(name, speed)
```

Set the playback speed of an animation, where 1 is the original speed forward, 2 would be twice as fast, and -1 would be reverse with original speed.

18.3.1.2 *Avatar*

The Avatar component is simple to use. It does not have functions to call as such, only an attribute called "appearanceRef" which should refer to the appearance JSON file that it should load. The attribute is set in the following (slightly convoluted) manner:

```
var ref = avatar.appearanceRef;
ref.ref = "YourAvatarFile.json";
avatar.appearanceRef = ref;
```

The change of this attribute automatically triggers the Avatar component to remove any existing Mesh component from its entity, and create new, once the JSON description has been loaded.

Once the Avatar component has finished loading the description, has instantiated the Mesh component(s) and their mesh assets have loaded, the character is ready to use for animation playback. At this point the Avatar component will emit a signal, avatarLoaded, to which application code can hook up.

18.3.1.3 *Character appearance description file format*

The description JSON file format is best described with an example. Here is an example of a multi-mesh object. The root-level mesh ("robot.json") is created into a Mesh component in the same entity as the Avatar component itself, while the "parts" or sub-objects are created into Mesh components in child entities. Note that the 3D object assets in this example, such as the "robot.json" are in Three.js inbuilt JSON mesh format, which contains vertex data, skeleton hierarchy and animation all in one file. For supported 3D file formats in WebTundra, see the 3D-UI documentation.

A 3D transform can be applied both to the main mesh and the sub-object meshes, where

- pos is a 3D vector translation
- rot is an Euler angle (degrees) rotation
- scale is a 3D vector scale

The sub-object meshes can also be parented to bones in the root level mesh's skeleton.

When such multi-part mesh is constructed, the AnimationController object will automatically drive animations in all of the sub-objects in a synchronized manner, given that the skeletons and animations in all the parts are comparible.

```
{
    "name"      : "RobotAvatar",
    "geometry"  : "robot.json",
    "transform" :
    {
        "pos": [0, 0, 0],
        "rot": [0, 0, 0],
        "scale": [1, 1, 1]
    },
    "materials" :
    [
        "submesh1_materialref",
        "submesh2_materialref"
    ],
    "parts" :
    [
        {
            "name"      : "Sword1",
            "geometry"  : "sword.json",
            "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [90, 0, 0],
                "scale": [1, 1, 1],
                "parentBone": "hand.L"
            },
            "materials" :
            [
                "submesh1_materialref",
                "submesh2_materialref"
            ]
```

```
        },
        {
            "name"      : "Sword2",
            "geometry"  : "sword.json",
            "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [90, 0, 0],
                "scale": [1, 1, 1],
                "parentBone": "hand.R"
            },
            "materials" :
            [
                "submesh1_materialref",
                "submesh2_materialref"
            ]
        },
        {
            "name"      : "Pants",
            "geometry"  : "robot_pants.json",
            "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [0, 0, 0],
                "scale": [1, 1, 1]
            }
        },
        {
            "name"      : "Hat",
            "geometry"  : "robot_hat.json",
            "transform" :
            {
                "pos": [0, 0, 0],
                "rot": [0, 0, 0],
                "scale": [1, 1, 1]
            }
        }
    ],
```

```
}
```

18.3.2 Examples

18.3.2.1 *Animation playback by using components directly*

This example script requires the robot.json mesh asset from the examples/Avatar directory of the WebTundra source tree. For the required script includes, look at examples/Avatar/index.html

```
var app = new Tundra.Application();
app.start();
var scene = app.dataConnection.scene;

var ent = scene.createEntity(0);
ent.createComponent(0, "Placeable");
ent.placeable.setPosition(0, 0, -5);

ent.createComponent(0, "Mesh");
var meshRef = ent.mesh.meshRef;
meshRef.ref = "robot.json";
ent.mesh.meshRef = meshRef;

ent.createComponent(0, "AnimationController");

ent.mesh.meshAssetReady.add(function(){
ent.animationController.play("Walk", 0.0, true, true); });
```

18.3.2.2 *Character instantiation using a character description file, and playing an animation*

This script requires all the assets from the examples/Avatar directory. Similarly to the example above, look at examples/Avatar/index.html for the needed script includes.

```
var app = new Tundra.Application();
app.start();
var scene = app.dataConnection.scene;

var ent = scene.createEntity(0);
ent.createComponent(0, "Placeable");
ent.placeable.setPosition(0, 0, -5);

ent.createComponent(0, "Avatar");
var ref = ent.avatar.appearanceRef;
```

```
ref.ref = "Avatar1.json";
ent.avatar.appearanceRef = ref;


ent.avatar.avatarLoaded.add(function(){
ent.animationController.play("Run", 0.0, true, true); });
```

19 Interface Designer - User and Programmers Guide

You can find the content of this chapter as well in the wiki of fi-ware.

19.1 Introduction

This document describes the use of the reference implementation provided by the Interface Designer GE. The interface designer consist of four parts:

- Scene tree editor. Provides a tree-view of all available entities in a 3D scene.
- Entity-component editor. Provides further fine-tuning of the internals of entity components. (for example, modifying the x,y, and z 'position' vector axes of a 'transform' component)
- Transform gizmo. A 3D object that provides direct in-world translating / rotating / scaling an entity, represented as a 3D object that is always centered into the object of interest. It consist of three colored XYZ axes, three planes and a central box.
- Toolbar. A toolbar consisting of viewing the history of editing (undo / redo stack), creating primitives, quickly adding entities, toggling grid and axes helpers, as well as switching between scene and EC editor, and switching between transform gizmo modes.

19.1.1.1 *Background and Detail*

This User and Programmers Guide relates to the Interface Designer GE which is part of the Advanced Middleware and Web UI chapter. Please find more information about this Generic Enabler in the following Open Specification.

19.2 User guide

19.2.1.1 *Brief introduction about entities, components, and attributes*

19.2.1.1.1 *Entities*

An **entity** is any dynamic object within a 3D world; It is the "living being" of a 3D scene in which exists. It has unique ID number used for referencing said entity. In this particular case, entities can be empty, or contain **components** that will define their functionality, look, and behavior.

19.2.1.1.2 *Components*

Components consist of set of **attributes**, in which information about specific functionality is stored. Components also have unique ID and a type name. A component can be "static" or "dynamic". A **static** component has a pre-defined set of attributes, that the user cannot remove, or add new ones. The set of attributes is defined by the underlying client application, and given as such to the user. For example, if the "transform" attribute is removed from "Placeable", the component would be defunct, since it has the responsibility to define the entity position in the 3D scene. A **dynamic** component is completely user-defined, which means the user can add / remove attributes by will for application-specific purposes.

Some of the core component types include:

- Name: Adds a name to the entity.
- Placeable: Gives the entity a position / rotation / scale (further in the text altogether called "Transform") into the 3D scene, or toggles its visibility.
- Mesh: A collection of vertices, edges and faces that define the appearance of a 3D object. The mesh formats supported are defined by the underlying client application.
- Camera: Provides a first-view look to a 3D scene.

- Script: Provides scripting to the target entity, or the whole scene, that can further expand functionality of the scene as required by the author (for example, a script that moves the camera with the arrow keys on the keyboard)
- Dynamic: The fore-mentioned "user-defined" component.

In general, an entity that contains "Name" and "Placeable", will acquire a name and a 3D position in the scene, but not appearance. A "Mesh" component can be added and a valid reference (absolute path or URL) to a mesh format file as the "AssetReference" attribute so that the entity gets appearance and a position in the scene.

19.2.1.1.3 *Attributes*

Attributes are atomic or complex data types that store human-readable information to their parent components. The attributes have unique names (there cannot be two attributes with the same name) . Some of the attribute types include:

- Bool: Boolean attribute, that accepts "true" or "false" value;
- Int: Integer attribute, for integer numbers;
- UInt: Unsigned integer, for positive numbers only;
- String: A string attribute that stores text;
- Float: Floating-point numbers;
- Float2: A tuple of 2 floating-point numbers (x,y);
- Float3: A tuple of 3 floating-point numbers (x,y,z);
- Float4: A tuple of 4 floating-point numbers (x,y,z,w);
- Quat: A quaternion, technically same as Float4 (x,y,z,w);
- Color: Technically same as Float4, but made separately for better readability (red, green, blue, alpha);
- Transform: A complex data type consisting of three "Float3"-s named "position", "rotation" and "scale";
- EntityReference: A string attribute that accepts entity IDs or names, to be used as references for application-specific purposes;
- AssetReference: A reference (usually an absolute or relative path, or URL) to a mesh file, script file, material file, shader script, etc..
- AssetReferenceList: A list of AssetReferences.

Asset references

AssetReference is a specialized string attribute, that is used to reference an asset that is in the local file system, stored on cloud, or hosted on a FTP server, therefore it can be an absolute path or URL. The underlying system should take care for fetching and loading the referenced asset.

19.2.1.1.4 *Local, replicated, temporary*

Furthermore, the availability of entities and components to the server and other clients, can be defined as **local** or **replicated** (also known as **synchronized**), and **temporary** or persistent.

- A local entity or component is visible only to the current running client. Their creation, removal or changes are not sent to the server, and so other clients "do not know" about said entity or component. Changing means adding / removing components of the entity, and in case of component, changing the values of its attributes.
- A replicated (synchronized) entity or component is the opposite of local; When a replicated entity or component is created / removed / changed, the server is notified which will forward the information to all the clients.

- A temporary entity or component means that the entity or component will not be considered when the scene is saved to a file. This is useful in cases where entities are not meant to became part of a permanent scene, for example, an external script that takes care of creating an entity and using it for its own purposes that are usually short-term.

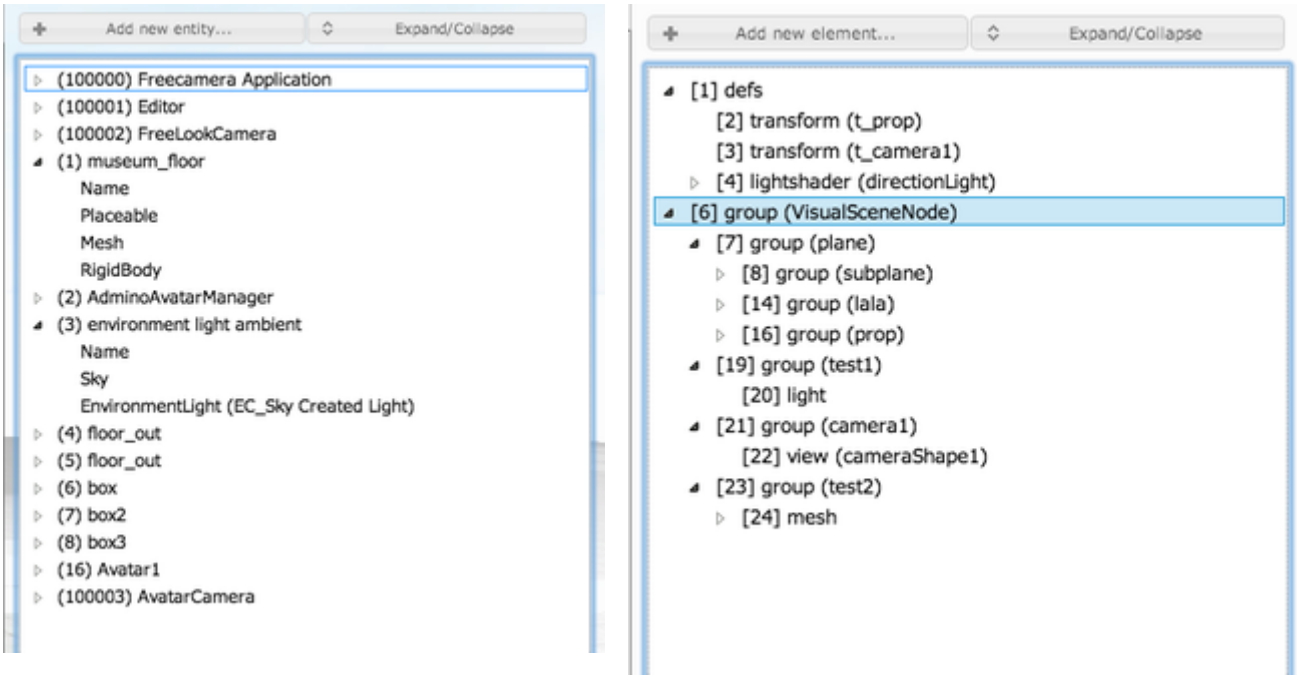
19.2.1.1.5 *XML3D vs ECA*

XML3D is compatible with the entity-component-attribute by defining its elements as both entities and components. A XML3D element can have child elements (equivalent to entity), while also having attributes (equivalent to component).

19.2.1.2 ***Scene tree editor***

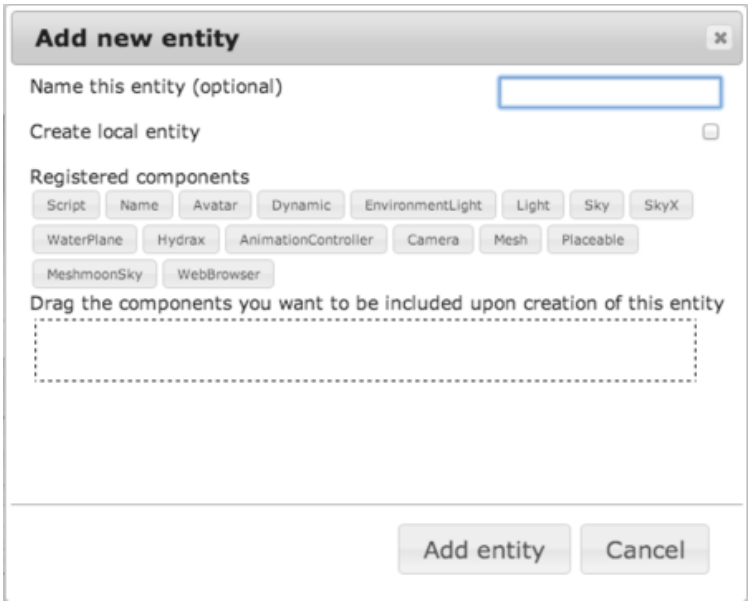
The scene tree editor consists of a tree-view widget that displays all available entities within a scene, and serves as the "head" of the interface designer, as it provides access to all available entities in a scene. The default shortcut for enabling the editor is **"Shift + S"**. Naming of the tree nodes is as follows:

- WebRocket
  - for entities **(id) Entity name**
  - for components **type_name (component_name)**
- XML3D
  - all elements: **[id] element_type (element_name)**

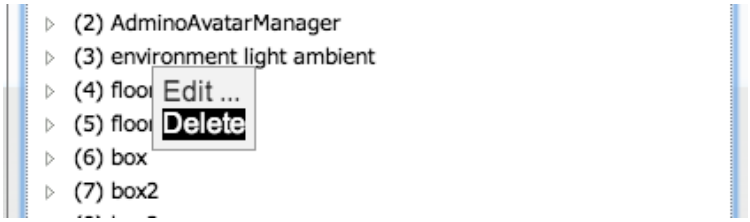


19.2.1.2.1 *Adding / removing entities and their components*

An new entity can be created by clicking on the "Add new entity" button, that will pop-up a dialog. Provide the name of the new entity (optional) and choose which components should be created with it by simply dragging the buttons to the marked area:



Removing an entity or a component from an entity, can be done by right-clicking of the desired entity / component, and selecting "Delete":



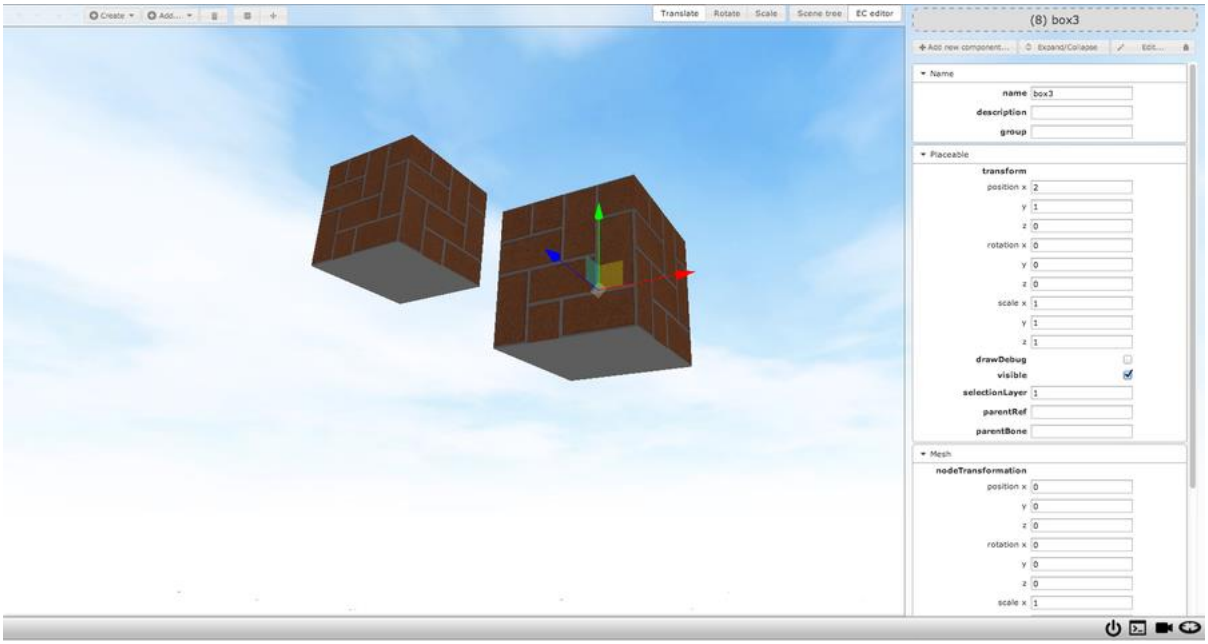
It will show a confirmation dialog:

19.2.1.2.2 *Editing individual entities*

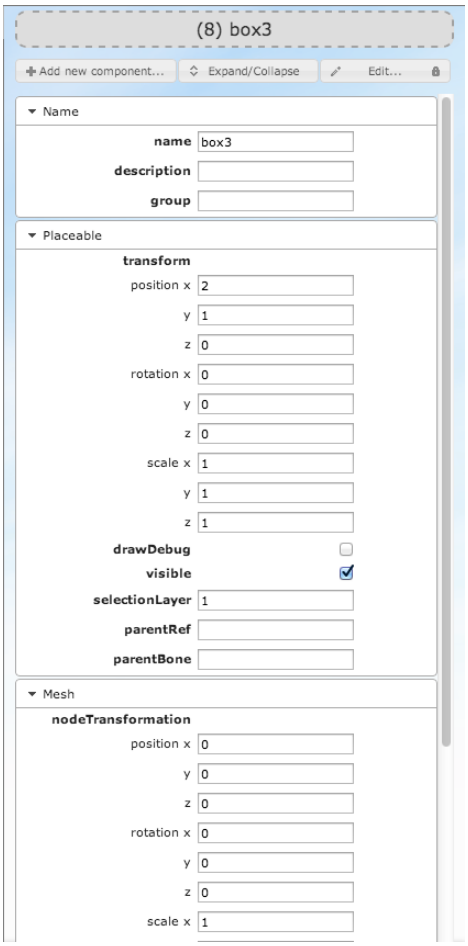
Right-clicking an entity from the tree-view and selecting "Edit..." in the context menu, will open up the Entity-component editor, that allows further fine-tuning of component attributes, such as "Mesh", "Transform" components etc. Right-clicking a component from the scene tree will also open the Entity-component editor, with the selected component as active (the accordion for the selected component will be expanded).

19.2.1.3 *Entity-component editor*

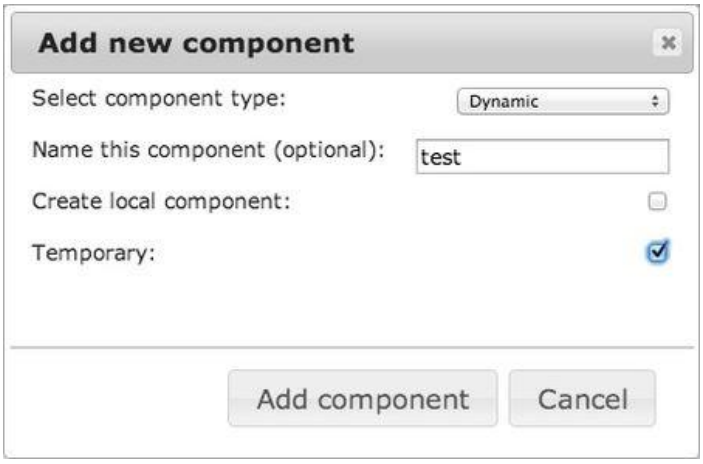
The entity-component editor further expands component and attribute information, placed in jquery-ui "accordions" which can be individually expanded / collapsed.



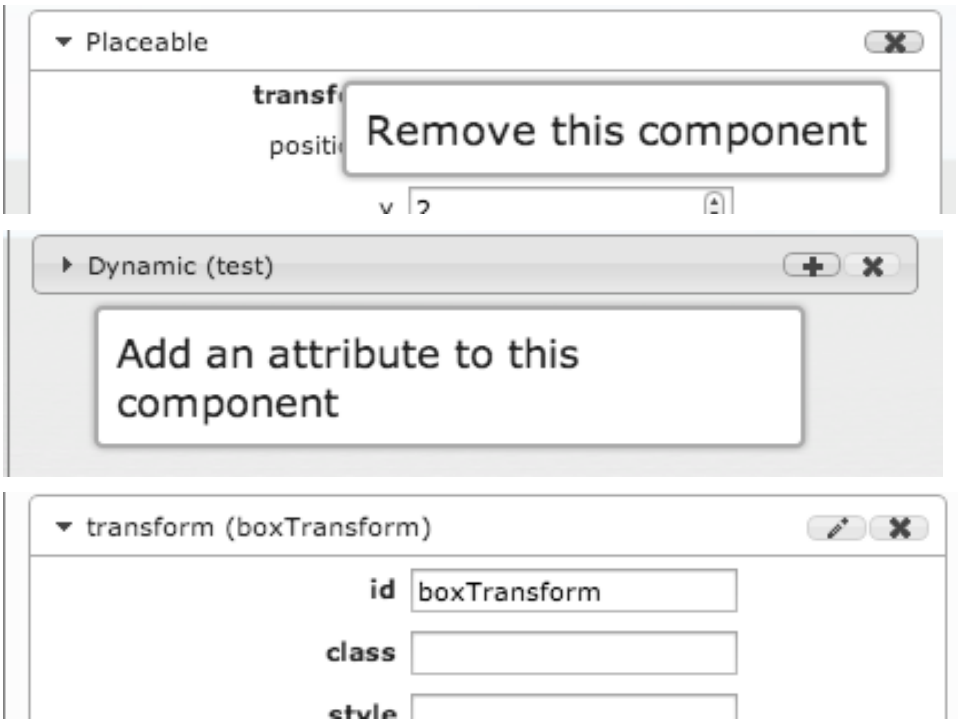
The editor will automatically show when an object has been selected, by right-clicking a tree item and selecting "Edit...", or by pressing "Shift + E" keyboard shortcut. It contains three additional buttons: **"Add new component / child element"**, **"Edit"**, and **"Expand/collapse"**



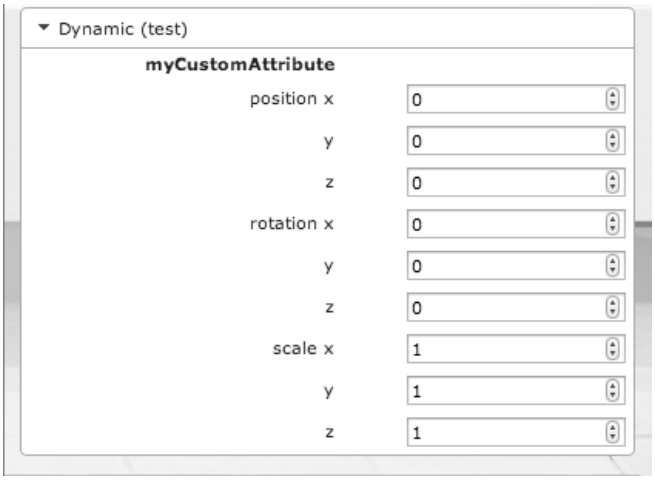
By right-clicking and "Edit" on an entity from the scene tree editor, or by clicking it directly on the scene, the entity-component editor will be shown, allowing of fine-tuning of attributes, as well as adding/removing of components. The header of the accordion contains the entity ID and entity name if specified in the following format: **"(id) <ENTITY NAME HERE>"**.
Pressing on **"Add new component"** will pop-up a modal dialog, asking for the component name, type chosen from a list of available components provided by the underlying client application (XML3D or Web Rocket), "Create local component" check-box, and "Temporary" check-box.



When the **"Edit"** button is toggled, the **"Remove this component"** (remove **"X"** icon) and **"Add new attribute"** (plus sign **"+"** icon) or **"Edit"** (pencil icon) buttons are shown on the accordion header where applicable.

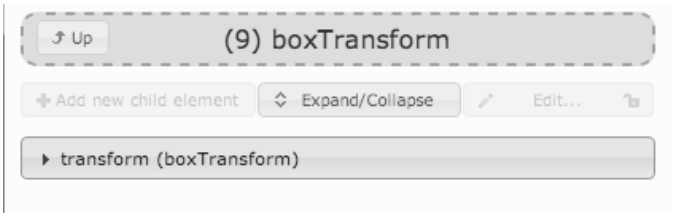


By pressing on the "Add new attribute" button, it will pop-up a modal dialog, asking for name and type of the attribute that is about to be created. The name cannot be duplicate of an existing attribute name. The type is chosen from a list, depending on the currently supported attributes by the underlying client application (XML3D or Web Rocket).



Pressing on "Remove this component" or "X" will pop-up a confirmation dialog, to prevent accidental removal.

Pressing on "Edit" in case of XML3D, will open up that child component into the editor. An "Up" button will appear on the left side of the label to return to the parent entity.



Pressing the "Expand/collapse" button will simply expand all or collapse all opened component accordions.

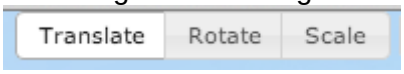
The attribute names and corresponding input boxes (check boxes in case of boolean attribute) are shown in a tabular view as the accordion body. Editing the attributes have immediate effect on the entity (no confirmation is needed).

19.2.1.4 *Transform gizmo*

The transform gizmo is always shown centered on the currently selected entity when the editor is enabled. It has three modes: translation, rotation and scaling. Each color represents the following:

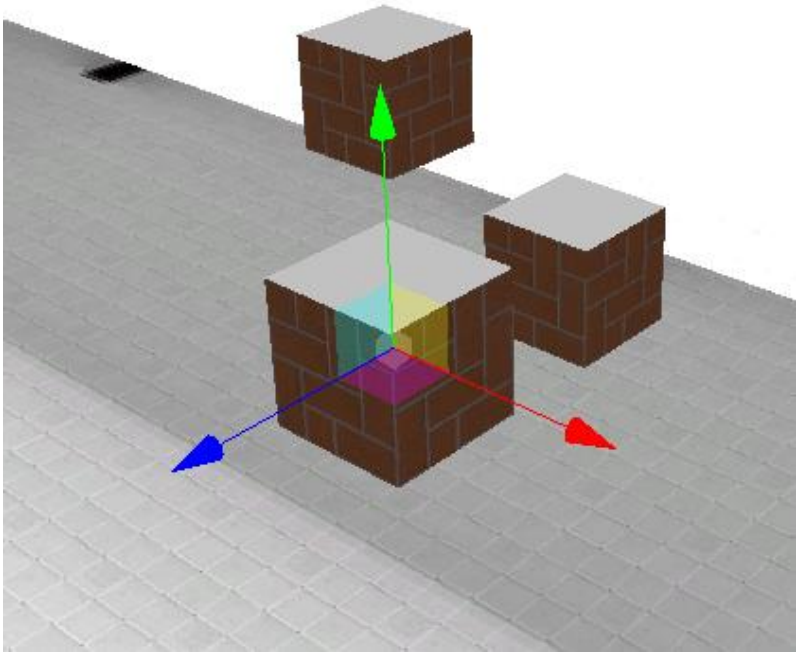
- X axis - red
- Y axis - green
- Z axis - blue
- XY plane - yellow
- YZ plane - cyan
- XZ plane - magenta

Switching between gizmo modes is done in the toolbar with three radio buttons:



19.2.1.4.1 *Translate*

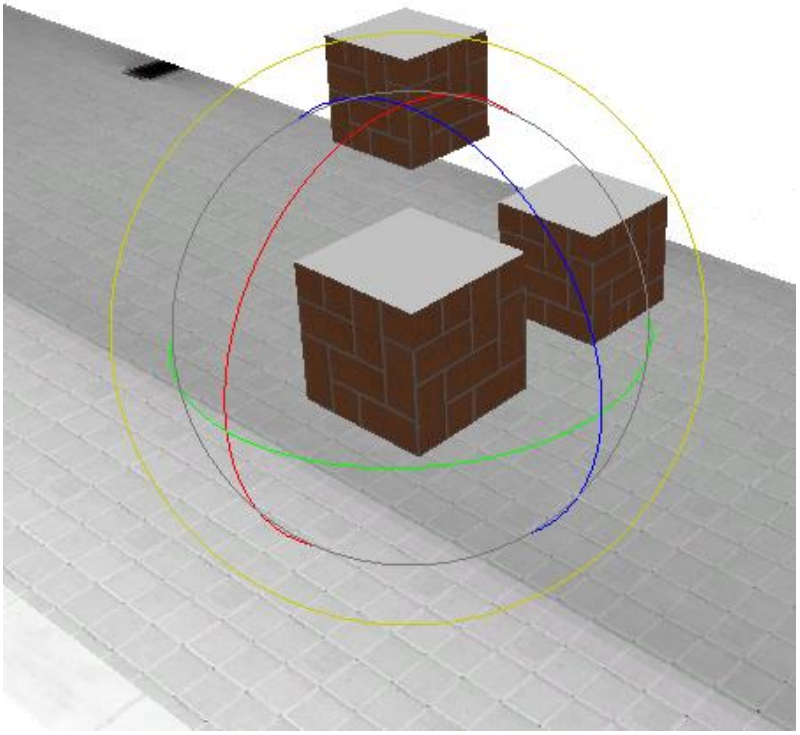
The **Translate** mode is used to change the position of the current 3D object. It consists of three arrows, three planes and one central box.



By dragging on the axes, the movement is restricted to a single coordinate, while dragging on the planes the movement is restricted to two coordinates. Dragging the central box allows free movement across the viewport.

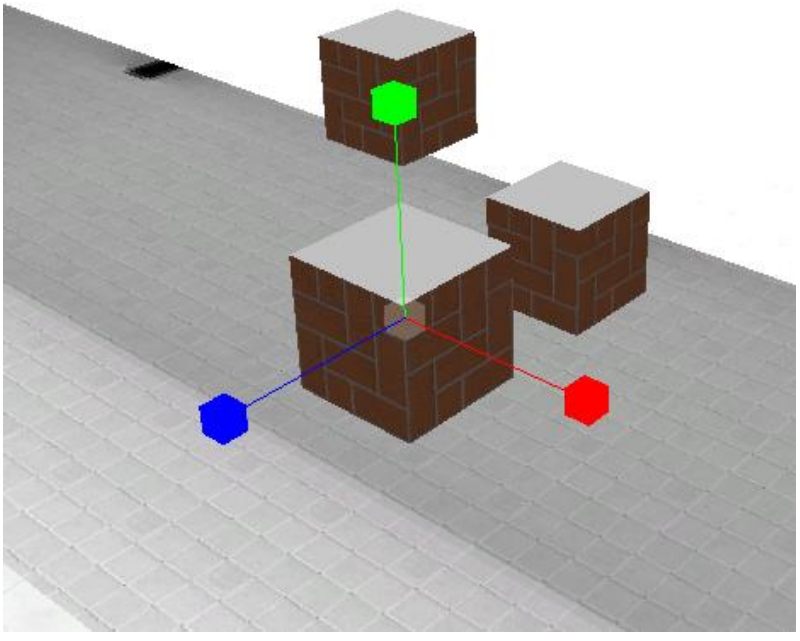
19.2.1.4.2 *Rotate*

- The "Rotate" mode is used to rotate the object in place. It consists of 3 arcs, each colored to the corresponding coordinate (as explained above), and 1 circle that allows rotation according to the current camera view.



19.2.1.4.3 *Scale*

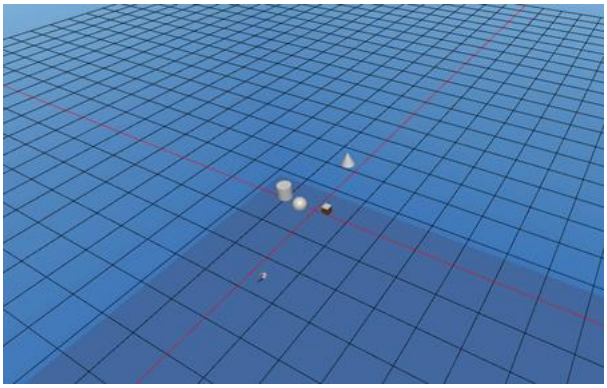
- The "Scale" mode is used to increase / decrease the size of the mesh. It consists of three lines with box on their tops and a central box. Dragging on the lines restricts the scaling to a single coordinate, while dragging on the central box scales all coordinates.

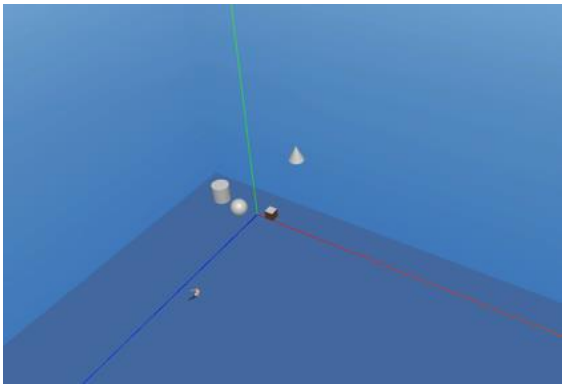


19.2.1.5 ***Toolbar***

The toolbar contains "Undo" and "Redo" buttons, "Create" button for primitives, "Add..." button for quick-adding empty entities, "Delete" button, "Toggle grid", "Toggle axes", "Translate | Rotate | Scale" radio buttons and "Scene tree | EC editor" radio buttons.

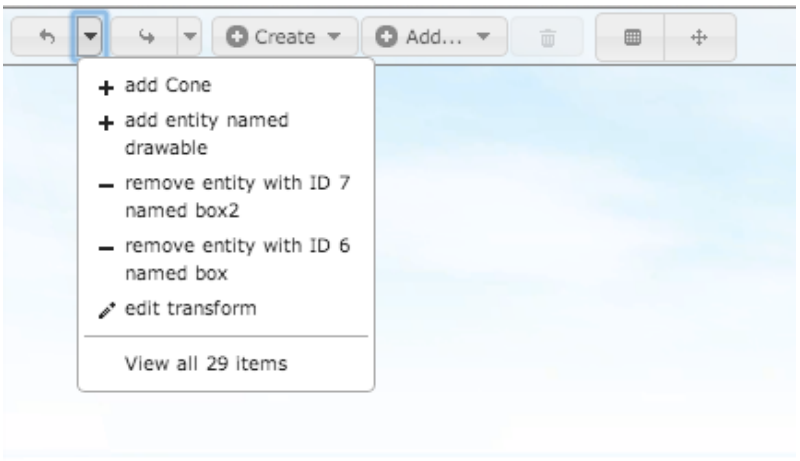
- Clicking on Undo will undo the last made edit / action. "Redo" repeats the last undone action.
- "Create" will pop-up a menu that has "Cube", "Ball", "Cone" and "Cylinder" items. Clicking on any of those will create a primitive object on the scene, depending on the selection.
- "Add..." will pop-up a menu that has "Movable", "Drawable" and "Script" items, that create an entity with the corresponding components. For example, a drawable entity is an entity that has a mesh component.
- Clicking on the "Delete" (trash can icon) button will remove the current edited object, for which it will ask with a confirmation dialog.
- "Toggle grid" (grid icon) toggles a grid in the 3D scene, with its center in the 0,0,0 point, and two crossed red lines for each X and Z axis.
- "Toggle axes" (arrows icon) toggles a simple axes mesh, similar to that of the gizmo, but serving only as a guide.
- "Translate | Rotate | Scale" serve for changing the transform gizmo mode. Disabled when no object is selected.
- "Scene tree | EC editor" switches between the scene tree and EC panels, as an alternative to the "Shift" + E keyboard shortcut.





19.2.1.5.1 *Undo / redo stack*

The undo stack keeps the last 50 edits made via the editor. Next to the "Undo" and "Redo" buttons there are much smaller buttons with arrow icon, that will pop-up a menu that shows 5 most recent actions. The items are always from most recent as the topmost item, to the least recent as the bottom item. For example, as shown on the following photo, selecting "edit transform" item, "undo" will be executed five times.



When the stack contains more than 5 items, there is an extra menu item "View all X items" that will show a window with all the actions that the stack keeps track of. Transversing through the actions is equivalent to clicking "Undo" or "Redo" multiple times.

Again, the topmost item is the most recent action, while the bottommost is the least recent action. The "Current state" item represents how deep into the undo stack the current state is. As the picture shows, there is one item above "Current state" and many below. Clicking on items above "Current state" and then "Ok" will call "redo" (as many times as it should) and clicking on the items below "Current state" will call "undo". If "Current state" is selected, it will have no effect.

19.3 Programmers guide

This GE is application-oriented, meaning that it is used as-is. The only programmer side of this GE is enabling / disabling the editor on demand, by using the methods provided in the Javascript library, and selecting an entity to be shown into the entity-component editor.