Programmers Manual for the GizmoBase Toolkit

Contents

1 GENERAL

1.1 What is Base or GizmoBase SDK

The GizmoBase library is the platform independent abstraction layer in the GizmoSDK family.

We always found out that we needed a platform abstraction API that isolated the different HW platform code solutions from our code so we could write once and run everywhere. We also found out that standard template library (STL) didn't work well in situations when we used shared libraries (.dlls), memory management was slow on Win32, exception management on PocketPC was not implemented, different programmers used different solutions on various platforms etc. so we tried to solve all these issues by a common layer API that forced many programmers in a group to write compatible code. The solution was a library that we call **GizmoBase.** The library is a C++ layer that implements most needed features for memory management, error management, templates, reference pointers, file I/O etc.

2 GIZMOBASE API OVERVIEW

The base library provides a platform abstraction layer in combination with a set of utilities for C++ programmers. The base library is fully platform independent allowing you to write advanced C++ code that runs on all supported platforms MacOS,Irix, Linux and Win32.

It covers utilities and design patterns like…

- *exception management*
  Classification of exceptions as notice, warnings or fatal exceptions with report mechanisms

- *observer/notifier pattern*
  Load balanced notification with priority queues and timeouts

- *templates*
  For dictionaries, lists, queues and dynamic arrays etc. in combination with smart pointers (shared reference management)

- *shared cloning*
  Recursive shared cloning of factories to build trees with selected shared node types

- *message and debug notifications*
  Report mechanisms and subscriptions

- *serialization*
  Persist storage etc. with serializing patterns

- *threads and mutexes*
  Multiprocess management

- *RTTI*
  Selected RunTimeTypeInformation with support for hierarchical/non hierarchical structures

- *Networking*
  Interfaces to TCP/IP, UDP using Ethernet or IRDA etc.

- *Utilities*
  Strings, File management, Environment settings, fast memory allocation, memory debugging etc.

3 GIZMOBASE API DETAILS

The purpose of the GizmoBase toolkit is to provide a generic software foundation that can be used on all supported platforms. The toolkit provides type definitions, templates and utilities that do not need to be changed between different platforms. You can use GizmoBase without the rest of the libs as a good foundation in your other software projects. You can find tutorials of GizmoBase in chapter 4 Tutorials.

3.1 Namespace or Prefix

As GizmoBase shall support platforms and compilers that lacks the support for namespace, we have chosen to use a prefix instead. The GizmoBase toolkit uses a prefix "gz" on all types, classes and global functions.

3.2 Defined types

The GizmoBase toolkit has its own definition of all C++ types used in the toolkit. It is based on the OpenGL definition. It can be used with or without any existing OpenGL headers.

See appendix Defined Types 7.3

3.3 Defined Macros

The following macros can be defined…

GZ_DEBUG	Define this in you debug build to get some debug features enabled. GZTRACE("xx") is defined and produces an output debug message.
GZ_MEM_DEBUG	Define this to get additional info in new/delete operators for memory dumps etc.
GZ_INSTRUMENT_CODE	Define this to get your performance macros instrumented

Platforms…

WIN32	Define this for Win32 builds or better to define GZ_WIN32
GZ_WIN32	Define this for Win32 PC builds
GZ_PPC_2002	Define this for PocketPC 2002 builds
GZ_MAC	Define this for Mac OS X builds

GZ_XBOX	Define this for Xbox builds
GZ_PS2	Define this for PS2 builds
GZ_LINUX	Define this for Linux builds
GZ_SGI	Define this for Irix builds

3.4 Key Registry Variables

GizmoSDK uses Key Registry variables to control user or global settings for the software. It works just like the windows registry on Win32 platforms or environment variables on Unix, but in a special GizmoSDK specific storage compatible between all platforms.

E.g. You can set a key TEST = 127 for a user. This variable is accessible from the GizmoSDK software just for this user. The value 127 is unique for this user. Another user can have the same variable name, but with a different value.

E.g. You can set a global key that gets accessible by all users reading the same value.

The following keys are defined…

<u>User Keys</u>

GIZMO_LANGUAGE	Current preferred language e.g. "SWE"
GIZMO_LANGUAGE_FILE	Current preferred language file. Default:gizmosdk.lng
GIZMO_PATH	A semi colon or comma separated path to gizmosdk files. E.g. "c:\textures,c:\bitmaps"
GIZMO_MODULES	
GIZMO_LICENSE	The license for the SDK.
GIZMO_CACHE_PATH	

<u>Global Keys</u>

3.5 Conversion Macros

gzPtr2Val(x)	Convert from memory pointer to value
gzVal2Ptr(x)	Convert from value to memory pointer
gzL(x)	Convert from string "" to wide string L""

3.6 Macro Constants

See appendix Defined Constants 7.4

3.7 Basic utilites & Templates

The GizmoBase toolkit adds some basic utilities and templates to the C++ family. The GizmoBase templates are similar to STL (Standard Template Library) templates except that they add functionality needed by the important gzReference interface (Reference management). The basic templates are: (*defined in gzTemplates.h*)

1. **gzList<class T>**
   Provides a fast linked list with pointers. Best usage for fast dynamic storage of pointers. Also adds sorted insertion to the list so you can add items to a list using sort values. Fast to add values at the end (insert) or at the beginning (pre_insert)

2. **gzListIterator<class T>**
   An iterator for gzList. Easy to use for iterating over all items in a gzList.

3. **gzArray<class T>**
   A static sized vector (array) class. Very fast for storage or retrieval of indexed values. The values are accessed by reference.

4. **gzDynamicArray<class T>**
   A dynamic sizeable vector class. Automatically adjusts size to index size.

5. **gzDict<class T1,class T2>**
   A dictionary for fast storage and retrieval of pointers using hash values. The T1 class is the lookup class and the T2 is the result pointer class.

6. **gzDictEntry<class T1,class T2>**
   An entry in a dictionary. The connection between the elements T1 and T2.

7. **gzDictIterator<class T1,class T2>**
   An iterator for gzDict.

8. **gzQueue<class T>**
   A FIFO or LIFO stack queue.

9. **gzDList<class T>**
   Provides a fast double linked list with pointers.

10. **gzDListIterator<class T>**
    An iterator for gzDList. Possible to iterate in both directions.

11. **gzAbs**
    Calculates the absolute value. Uses fast floating point arithmetic for float and double values.

12. **gzAbsMax**
    Calculates the maximum of absolute value.

13. **gzRelativeDiff**
    Calculates the relative diff of two values

14. **gzBaseClass<class T>**
    Used to subclass data (class T) to provide a class for basic primitives like gzFloat , gzLong etc. Use this in combination with gzRefData<> to hold reference based primitives.

15. **gzHashPair< class T1 , class T2>**
    Used to add hash evaluation to a class so it can be used in dictionaries etc.

16. **gzLateBindData< class T>**
    Used to add late binding of data. Data is allocated on first access. E.g. usage as a member variable in a class. The member variable is created at first usage and not at class instantiation.

17. **gzLateBindArray< class T>**
    Used to add late binding of array data

18. **gzMatrix< class T>**
    Matrix storage of class T.

19. **gzDataSort< class T >**
    Hyper fast insertion sort. Invented by Anders Modén.

If a template enters an error condition an exception of type *gzFatalError* is thrown. See gzExceptions.h

3.8 Basic Utility classes

The basic utility classes are: (*defined in gzBase.h*)

a. **gzString**
   A string class that encapsulates C strings in a controlled manner.

b. **gzTime**
   System time management. You do not need to be superuser to set this time. Accuracy down to 1/1000000 of a second. Y2K compliant.

c. **gzRandom**
   A random number generator that gives values between 0.0 and 1.0.

d. **isPowerOfTwo**
   True if number is x^2

e. **getUpperPowerOfTwo**
Rounds a number to nearest upper number that is a power of 2

f. **getUpperPowerOfTwoExponent**
Same as above but the power number

g. **gzLtoA**
Converts an integer to an ASCII representation

h. **gzDtoA**
Converts a double number to an ASCII representation

i. **gzFaculty**
Calculates the faculty n!=1*2*3*4*5*6…*(n-1)*n

j. **gzCombination**
Calculates the combination. Se you math handbook!

k. **gzPrecision**
Rounds of values to a given precision

l. **gzUTF8ToULong**
Converts a UTF8 character encoding to a ULong.

m. **gzULongToUTF8**
The reverse operation of previous function.

n. **gzUTF8Size**
Calculates the size of storage of UTF8 encodings.

o. **gzKeyDatabase**
Global and user keys with access control. Keys are used to set up paths and licenses etc. The following example keys are defined:

GIZMO_PATH – used to define paths used in search for files by the engine. Multiple paths are separated with semicolon e.g. "c:\GIZMO\files\ **;** c:\GIZMO\databases\"

GIZMO_LICENSE – used to enter license key e.g. "7348923345666534234437"

GIZMO_MODULES – path to add files for dynamic loading. Separate files with semicolon e.g. "test.dll **;** exec.dll"

p. **gzDirectory, gzDirectoryEntry, gzDirectoryIterator**
Directory management classes. Use this to traverse a directory structure.

q. **gzNameInterface**
GizmoBase class for naming of objects

r. **gzUpdateIDInterface**
GizmoBase class for adding unique update id to a class.

3.9 Memory Management

Some GizmoBase classes are derived from the *gzSharedMemory* base class. This is important because GizmoBase uses reference management. An object allocated in one code module can be deleted by another code module. If they do not share the same memory manager, there is a great potential risk to get corrupt memory heaps or even software crashes. So if you write code that is shared between .dll/.so libraries etc, you should try to use *gzReference* as a base class if the data is referenced or *gzSharedMemory* if the data is not.

When using MFC, it is a good rule to add the GizmoBase includes directives (#include …) after all your normal MFC includes. If you get problems with the std namespace you can try to add the gizmo header files just after the include of "stdafx.h".

The GizmoBase libraries can be compiled with the "GZ_MEM_DEBUG" directive. This will enforce all memory allocations from both your program and GizmoBase to be reported. This option is only available for source code customers.

If you define "GZ_MEM_DEBUG" before inclusion of GizmoBase headers, you will get memory-debugging features enabled for your own code.

You control the memory behavior through the ***gzMemoryControl*** class.

By calling ***gzMemoryControl:: traceAlloc(TRUE)*** you will get allocation trace of memory if you have enabled "GZ_MEM_DEBUG". If you call ***gzMemoryControl::dumpAllocMem()*** you will get a dump of current allocation status. At the end of execution you will also get a lost memory report.

Eg. An output from a program

```
GizmoSDK Memory::--!! (Thread:0xb48) Malloc 24 bytes at 0xdc5c10 (index:0) (count:0) (tot:24)
->File: x:\gizmosdk\gizmobase\examples\smartpointer\smart.cpp  Line: 76  Ver: 1.2

GizmoSDK Memory::--!! (Thread:0xb48) Malloc 28 bytes at 0xddc440 (index:2) (count:17) (tot:68)
->File: x:\gizmosdk\gizmobase\examples\smartpointer\smart.cpp  Line: 84  Ver: 1.2

GizmoSDK Memory::--!! (Thread:0xb48) Free 28 bytes at 0xddc440 (index:2) (count:17) (tot:40) -
>File: x:\gizmosdk\gizmobase\examples\smartpointer\smart.cpp  Line: 84  Ver: 1.2

GizmoSDK Memory::--!! (Thread:0xb48) Free 24 bytes at 0xdc5c10 (index:0) (count:0) (tot:16) -
>File: x:\gizmosdk\gizmobase\examples\smartpointer\smart.cpp  Line: 76  Ver: 1.2

GizmoSDK Memory::-- GizmoSDK Lost Memory Report ---------------------------------------------

GizmoSDK Memory::!! (thread:0xb48) Lost mem 16 bytes at 0xdc5c58 (count:1) (tot:16) (state:1)

GizmoSDK Memory::-- End of Lost Memory Report ---------------------------------------------
```

The index number is the total number of present allocations made at the malloc time. The count number is the number of total allocations made at the malloc time. The tot number is the total number of bytes allocated at the time of the malloc or free.

Note that GizmoBase always has a default message receiver so you will get 16 bytes lost if you have used any message functions.

If you want to check all memory allocations without using the "GZ_MEM_DEBUG" define, you can use ***gzMemoryControl::debugMem(TRUE).*** This will enable memory trace of all allocation even if the "GZ_MEM_DEBUG" is not defined. You will not get line info etc as you get from "GZ_MEM_DEBUG" defined news and deletes, but you will get allocations and frees etc.

You can get the number of currently traced allocations with

gzMemoryControl:: getTracedAllocMem();

The GizmoBase memory management also defines a fast memory pool for 16 byte allocations (commonly used in Gizmo3D). This feature "eats" up a chunk of memory (16 MByte) and then uses its own allocation scheme for this. You can enable this by calling ***gzMemoryControl::enableFastMemory()***.

3.10 Debug & Message Support

You can use the macro GZTRACE to output "printf" formatted debug output. The output is only active when the macro "GZ_DEBUG" is defined. If the macro is not defined, the GZTRACE macro is replaced by a null statement.

e.g. ***GZTRACE("My debug message %d",36);***

You can also add trace info by adding the macro GZ_DEBUG_INFO("version string"). This macro creates a string with line number, file name etc that can be used to trace the output.

e.g. ***GZTRACE(" Some debug info in %s", GZ_DEBUG_INFO(GZ_VERSION_STR)*** will output "Some debug info in File: test.cpp Line: 123 Ver: 0.99 B 19". You may also use your own version string info to reflect your own software version.

Sometimes you want to preserve your debug information and therefore there is a built in message system that can report different levels of information. The basic levels are defined as

```
typedef enum {
                        GZ_MESSAGE_MEM_DEBUG    =0x1000 ,
                        GZ_MESSAGE_PERF_DEBUG   =0x1001 ,
                        GZ_MESSAGE_DEBUG        =0x2000 ,
                        GZ_MESSAGE_NOTICE       =0x3000 ,
                        GZ_MESSAGE_WARNING      =0x4000 ,
                        GZ_MESSAGE_FATAL        =0x5000 ,
                        GZ_MESSAGE_ASSERT       =0x6000 ,
                        GZ_MESSAGE_ALWAYS       =0x7000

            } gzMessageLevel;
```

Default message level is GZ_MESSAGE_NOTICE. You can add your own levels based on these defined enums. When the reporting level is set to debug (0x2000), all messages with a higher or equal level to (0x2000), is routed to the default output message receiver.

A typical usage is to add the following code snippet to send debug messages..

```
GZMESSAGE("Error sender",GZ_MESSAGE_DEBUG,"Something went wrong");
```

This can be dynamically activated in your release software to send debug info etc. e.g. in a debugging mode.

You can add your own message receiver to the code to catch the messages. You might want to catch the messages and print them in your own window etc. The default message receiver prints to the console output (stderr). The first user initiated receiver replaces the system default handler. See example 5.3 how to set up the popular ATLTRACE macro on a Win32 platform to use output in your compiler.

Sometimes you wish to log all messages in a session. You can then instantiate a gzLogger. This will catch all messages and route them to your selected output. See 3.20 Serializing data for more info about various output techniques.

```
// when this instance is created it will catch all messages.
gzLogger logger("mylog.txt");

// When the instance is deleted it will write the messages to the actual output
```

If you wish to store all messages in a database that you can work with you can use the ***gzMessageDatabase*** class. Typical usage is to use ***GZ_ASSERT*** in combination with levels and modules so you can store the messages per level and per sender. This way you can e.g. create test cases that are built into the code, when "GZ_DEBUG" is defined.

e.g.

```
// Lets say you want to exercise your code with some tests

// Create a message database

gzMessageDatabase db;

. . .

// Exercise some code modules

. . .

// Lets say this particular modelu does a test

GZ_MODULE_LEVEL_ASSERT_TEXT("My Test Module",GZ_MESSAGE_WARNING,getLevel()>2.0
,"The water level is to low");


// When the instance is deleted it will write the messages to the actual output
```

3.11 Assertions

You can use ***GZ_ASSERT(test)*** macro to do debug assert tests. If **test** is evaluated to false, it will report the code module name, line number and the actual test. If the standard message receiver is installed (no user defined installed), the assert will also optionally terminate the program.

The assert will be removed if not "GZ_DEBUG" is defined. Normally active in debug builds and absent in release builds.

You can also assert with a text output. ***GZ_ASSERT_TEXT(test, text)*** will assert if **test** fails and output the **text**.

Sometimes you might want to trace or test a code module using asserts. You can then route various assert to different error levels by using ***GZ_LEVEL_ASSERT(level, test)***. This will assert message on level **level** if **test** fails. The level assert will not terminate the program.

E.g. to do a check if a certain level is ok, the ***GZ_LEVEL_ASSERT_TEXT(level, test, text)*** can be used. In the case of faulty level, the message is sent to a logger using the warning level.

```
GZ_LEVEL_ASSERT_TEXT(GZ_MESSAGE_WARNING,getLevel()>=2.0,"Water level is less than
two meters");
```

You can also use asserts with a sender (module) id. ***GZ_MODULE_ASSERT(module,test)*** will assert from sender **module** if **test** fails.

3.12 Performance Measuring

In GizmoBase you will find support to benchmark your code and find bottlenecks and optimization possibilities. You can do both simple timing as well as hierarchical analysis of recursive called functions.

To do a simple time and frequency measuring you can use the Timing macros. By using ***GZ_TIMER_START(name)*** a timer called **name** is started. To stop the timer ***GZ_TIMER_STOP(name, minTime)*** is called. If the total time is larger than the value specified with **minTime** the time is reported with a message. To get a message with the exact processing time **minTime** is assigned the value 0. It is possible to have several timers in the same code. Every timer has a unique name.

```
GZ_TIMER_START(timer1);         // provide a name for the timer

// … code to test …

GZ_TIMER_STOP(timer1,0.75);     // Stop timer "timer1" and if diff was >=0.75
                                // report a message
```

To do more complicated performance measuring you shall use the *gzPerformance* class. By using the *gzPerformance*, information about the computer performance can be monitored. For instance, a window will be opened containing real time printouts with percentages of the performance information etc.

3.12.1 Collecting the performance data

There are four different approaches to receive performance information, either by calling the functions using or not using a defined macro or by implementing a body using or not using defined macros.

By calling the functions using a defined macro the performance can be enabled or disabled by using the #define. To do this you include the following in your header file:

```
#define GZ_INSTRUMENT_CODE    // must be defined before
                              // include of "gzPerformance.h"
#include "gzPerformance.h"
```

Include this block in your performance source file:

```
GZ_ENTER_PERFORMANCE_SECTION("Objects and Events");
CreateManyObjects();
SendManyEvents();
GZ_LEAVE_PERFORMANCE_SECTION;
```

Note that these examples will be dependant on the definition of GZ_INSTRUMENT_CODE.

Another way is to not use a defined macro when calling the functions. The performance will always be active and not possible to disable as when using a defined macro. Then include the following line in the header file:

```
#include "gzPerformance.h"
```

Include this block in your performance source file:

```
gzEnterPerformanceSection("Objects and Events");
CreateManyObjects();
SendManyEvents();
gzLeavePerformanceSection();
```

It is also possible to implement a body using defined macros. Then this define is included in your header file:

```
#define GZ_INSTRUMENT_CODE    // must be defined before
                              // include of "gzPerformance.h"
#include "gzPerformance.h"
```

Include this block in your source file:

```
{
     GZ_INSTRUMENT_AUTO;    // Automatically defines a scope body
     CreateManyObjects();
     SendManyEvents();
}
```

Or include this block if you want to name it yourself:

```
{
     GZ_INSTRUMENT_NAME("Objects and Events");    // named body
     CreateManyObjects();
     SendManyEvents();
}
```

Note that these examples will be dependant on the definition of GZ_INSTRUMENT_CODE.

The last approach to receive performance information is to use a body without defined macros. If you are using a body, the performance data is created for the whole function. This is preferable if you don't know when the calls return from the function, compare with GZ_BODYGUARD.

Include this block in your header file:

```
#include "gzPerformance.h"
```

Include this block in your performance source file with a scope:

```
{
    gzPerformanceBody PerfMon("Objects and Events");
    CreateManyObjects();
    SendManyEvents();
}
// This way the performance data is collected properly even if exceptions are
thrown or the function returns within the scope.
```

3.12.2 What kind of performance data is available

The following code is an example of how you can do a performance test:

```
     gzMessage::setMessageLevel(GZ_MESSAGE_MEM_DEBUG);

     gzStartPerformanceThread();     // Start the main thread performance timing

     gzSleep(GZ_SLEEP_SECOND);       // Sleep one second
     {

          gzEnterPerformanceSection("test1");  // Enter the first code section

          gzSleep(GZ_SLEEP_SECOND*2);          // Sleep two seconds

          {
               gzEnterPerformanceSection("test2");  // Enter the second section

               gzSleep(GZ_SLEEP_SECOND*2);          // Sleep two seconds

               gzLeavePerformanceSection();
          }


          {
               gzEnterPerformanceSection("test2");  // Enter the second section
```

```
                gzSleep(GZ_SLEEP_SECOND*3);          // Sleep three seconds

                gzLeavePerformanceSection();
            }


            gzLeavePerformanceSection();
        }

        gzStopPerformanceThread();

        gzDumpPerformanceInfo(GZ_PERF_DUMP_ALL);
```

The above section gives the output below. The collected data is based on the time of execution. As the output below displays, one can see the ID of the thread, total execution time, and all accumulated and hierarchical sections for all running and stopped threads. One can see section name, execution time and percentage of the total thread time, iterations, average execution time and callers for all sections.

```
----------- Performance Dump (2003-11-21 10:25:47) ------------

- Running threads

- Stopped threads

  - Thread ID: 0x320
    Exec Time: 8 seconds

    - Accumulated sections

      - Section       : test1
        Exec Time     : 6.99991738 seconds (87.4994 %)
        Iterations    : 1 (rec:0)
        Avg Exec Time : 6.99991738 seconds (87.4994 %)
        Callers       : 1

      - Section       : test2
        Exec Time     : 4.99990975 seconds (62.4992 %)
        Iterations    : 2 (rec:0)
        Avg Exec Time : 2.49995487 seconds (31.2496 %)
        Callers       : 1

    - Hierarchical sections

      - Section       : test1
        Exec Time     : 6.99991738 seconds (100 %) (tot 87.4994 %)
        Iterations    : 1
        Avg Exec Time : 6.99991738 seconds (100 %) (tot 87.4994 %)

        - Section       : test2
          Exec Time     : 4.99990975 seconds (71.4281 %) (tot 62.4992 %)
          Iterations    : 2
          Avg Exec Time : 2.49995487 seconds (35.7140 %) (tot 31.2496 %)
-------------------- End Performance Dump ---------------------
```

As you can see the total time spent in the execution thread (between start/stop performance thread) is 8 seconds. Section “test1” has been called 1 time ands section “test2” has been called 2 times by the same parent section (“test1”).

In the accumulated section we can see the sum of all fears. The percentage in this case is how much execution time of the overall thread execution time was spent in the section. E.g section "test2" spent total 5 seconds of 8 seconds total = 5/8 = 62.5 %

In the hierarchical sections we can see that section "test2" was called by section "test1" and that section "test2" used 71.4 % of its parent (section "test1").

If you want a programmatic control of the section results for "test2", you can retrieve the performance info with the method ***gzGetSectionResult("test2")***

To add memory data, just include GZMESSAGE with GZ_MESSAGE_MEM_DEBUG as message level as below:

```
// Enable Memory debugging
gzMemoryControl::debugMem(GZ_TRUE);
gzULongLong nMemUsage = 0;

// Get number of bytes that is allocated and traced in use
nMemUsage = gzMemoryControl::getAllocMem();

// Add information about the Memory allocation
gzMessage::setMessageLevel(GZ_MESSAGE_MEM_DEBUG);

const char* sTime = gzTime::now().asString();
GZMESSAGE(GZ_MESSAGE_MEM_DEBUG, "---- Memory Dump (%s) ------", sTime);

GZMESSAGE(GZ_MESSAGE_MEM_DEBUG, "----- Mem Usage     : %d", nMemUsage);

GZMESSAGE(GZ_MESSAGE_MEM_DEBUG, "------------ End Memory Dump ------");
```

Don't forget to reset the message level before the next gzDumpPerformanceInfo.

3.12.3 Dumping the information in a window

To get the window with the performance information an additional function can be used which dumps hierarchical information for both terminated and active threads.

Include this block in your header file:

```
#include "gzPerformance.h"
```

Include this block in the constructor of your performance source file or perhaps in an OnKeyDown function:

```
gzMessage::setMessageLevel(GZ_MESSAGE_PERF_DEBUG);

gzDumpPerformanceInfo(GZ_PERF_DUMP_ALL);
```

Or you can use the automatically graphic output where you use the gzWindow with a gzApplication:

```
gzWindow *win = gzCreatePerformanceWindow(const gzArray<gzString> & sSections,
gzULong nIterations = 100);
```

To be able to do this you have to use Gizmo3D. You can read more about windows and applications in the Gizmo3D programmers manual.

3.13 Reference Management

The Gizmo3D scene graph uses reference managed data. This is basically a base class for memory with a ref() and an unref() method. The ref() method increments a reference counter starting from zero at creation and the method unref() decrements the counter. As the counter reaches zero on a unref(), the last user of the shared memory has released its handle to the memory and the memory gets deleted.

You can create data with

```
class MyRefClass : public gzReference ….
```

and then do

```
MyRefClass *data=new MyRefClass;
```

You can immediately release the data by a normal delete, but when you have used the methods ref() and unref() you should NOT use delete.

The reference-managed class can be used with a number of utilities that all uses shared references and ref/unref. E.g. gzRefList or the smart pointer gzRefPointer templates. A typical situation is when you want to share data between multiple lists. By deriving your data from the gzReference class and use gzRefLists, you do not need to keep track of how many items are shared etc. when an item is removed from all lists, the item is automatically destroyed.

You can decrement a reference without automatic destruction using the unrefNoDelete(). It decrements the reference counter, but leaves the instance undeleted when the counter reaches zero. You can call checkDelete() upon such instances to check if the reference count is zero and then delete the instance if the counter was zero.

3.14 Language translations & support

GizmoBase supports language translation by default in the GZMESSAGE macro. The message gets translated in a language dependant lookup table. If the table is empty or the message that you send doesn't exist in the table, the system adds the message to the table and let you later enter a correct translation for this message.

The message lookup table is by default saved in the UNICODE16 text file "gizmosdk.lng". This file can be edited by WordPad etc. You can change the file name by setting the environment variable GIZMO_LANGUAGE_FILE to another file name.

To enable language translation in your software, you need to activate language support with the command

gzMessageTranslatorInterface::useDefaultTranslator(TRUE)

You set the preferred language either by setting the environment variable GIZMO_LANGUAGE to a text string, or set the language with the command

gzMessageTranslatorInterface::setCurrentLanguage("YourLanguage")

To do a quick translation of a string you can use the GZ_TR macro.

Just do a gzString trans= GZ_TR("test") to do a translation lookup of the "test" string.

It is possible to install your own translator. Derive from the ***gzMessageTranslatorInterface*** and install your virtual methods to do your custom translations.

3.15 Time management

gzTime etc..
XXX

3.16 Dynamic Data

A very powerful mechanism in GizmoBase is the usage of dynamic data or Dynamic Types. The situation often arise when the Scene Graph user wants to add some kind of data to a node or to a file format, but the actual type was not know when the programmer wrote the scene graph. The user must then be able to add unknown data types to the scene graph. E.g he wants to add a latitude/longitude coordinate pair to the graph. He could in that case use *gzDynamicType* data.

The base class *gzDynamicType* contains a number of pre defined types. They are

GZ_DYNAMIC_TYPE_STRING	A gzString value
GZ_DYNAMIC_TYPE_VOID	An unitialised value
GZ_DYNAMIC_TYPE_NUMBER	A real number
GZ_DYNAMIC_TYPE_POINTER	A pointer (char *)
GZ_DYNAMIC_TYPE_REFERENCE	A reference pointer gzReference *
GZ_DYNAMIC_TYPE_ERROR	An error value
GZ_DYNAMIC_TYPE_ARRAY	An array
GZ_DYNAMIC_TYPE_VEC3	A gzVec3 value

By supplying such values in the constructor of the *gzDynamicType* you can initialize the *gzDynamicType* automatically. When you wish to retrieve data, you can use the getXXX methods to get the actual data. The available methods are *getString(), getNumber(), getPointer(), getReference(),*and *getVec3()*. E.g. *getNumber()* returns the number.

XXX explain how to retrieve data of the types void, error, array

```
gzDynamicType a = 10 ;

gzDouble b=a.getNumber();

gzDynamicType c = gzString("Nisse");
```

```
gzString d=c.getString();
```

If you try to get a value from a dynamic type that was initialized with another type it will throw an exception. E.g. You can not set a number and use *getString()*.

The function *getDynamicType()* delivers a string which describe the type. The strings are:

GZ_DYNAMIC_TYPE_STRING	str
GZ_DYNAMIC_TYPE_VOID	void
GZ_DYNAMIC_TYPE_NUMBER	num
GZ_DYNAMIC_TYPE_POINTER	ptr
GZ_DYNAMIC_TYPE_REFERENCE	ref
GZ_DYNAMIC_TYPE_ERROR	error
GZ_DYNAMIC_TYPE_ARRAY	array
GZ_DYNAMIC_TYPE_VEC3	vec3

If you want more complex types you can build your own data types with the *gzDynamicTypeCustom*<> template. This way you can add any type of structure to dynamic type data. The type has to be derived from *gzSerializeData* and have a *getDataTag()* method. To get the type of a custom type the *getDataTag()* is used instead of *getDynamicType()* which is used for built-in types.

```
// Create data

gzDynamicType data;

gzAttribute_LatPos   lp;        // Must be derived from gzSerializeData and have a
                                // getDataTag() method

lp.latitude=1.008;
lp.longitude=0.2345;
lp.altitude=1000;

data=gzDynamicTypeCustom<gzAttribute_LatPos>(lp);    // Creates the dynamic data

// use the data

if(data.is(xxx) for the normal builtin types defined in gzDynamic.h
{
}
else if(data.is(gzAttribute_LatPos::getDataTag()))   // Check for specific type
{
     lp=gzDynamic_Cast<gzAttribute_LatPos>(data);
}
```

We will later se how to add dynamic data to objects etc.

3.17 Objects & User data

The purpose of the *gzObject* class is to be the base of all data that needs some kind of automatic serialization and that has a defined set of methods with inheritance etc. If you should construct a c++ object and need to have the RTTI mechanism, reference management, associated data and you want to be able to save this special c++ object with an adapter, you should start with the *gzObject* as the base class of your c++ object.

3.18 Threads

The GizmoBase toolkit libs are fully multithreaded. They use the class *gzThread* or *gzThreadTicker* as base classes. The *gzThread* class creates a thread and starts the virtual method ***process()***. The *gzThreadTicker* reuses a thread pool so all tickers that shares a pool ID uses the same thread. They must no lock the execution because they will then lock out the rest of the tickers in the same pool.

A typical sample of a use thread looks like this.

```
class MyThread : public gzThread
{
public:
        MyThread(){};

        virtual ~MyThread(){};

        virtual gzVoid process()
        {
                while(!isStopping())     // check if stop is requested
                {
                        // do some stuff
                }
        }
};
```

By calling ***MyThread *pThread=new MyThread;*** and then start execution by ***pThread->run()*** it will run until you call ***pThread->stop();***

The usage of a *gzThreadTicker* works like this.

```
class MyThread : public gzThreadTicker
{
public:
        MyThread(gzULong pool):gzThreadTicker(pool){};

        virtual ~MyThread(){};

        virtual gzVoid onTick()
        {
                // do some stuff
        }
};
```

You instantiate it by providing an id that represents the pool id (one reused thread). E.g ***MyThread *pThread=new MyThread(99);*** and then start it by setting the tick interval (how often it shall be called) ***pThread->setTickInterval(0.1);***

Remember that if you block the ***onTick()*** method you will block all other tickers in the same pool.

3.19 Plugins

GizmoBase can load platform independent written code as plugins. GizmoBase can not use the same plugin between platforms, but you have to compile a plugin for a specific platform.

GizmoBase will have script plugins in the future that can be fully platform independent. Code written in C# or in Java are good candidates.

Se plugin sample 0 Dynamic Plugin

Remember that GizmoBase uses the environment variable "GIZMO_MODULES" to automatically load plugins. Plugins that are configured in this environment variable will be automatically loaded when the program execute ***loadModules()***.

```
e.g. gzSetEnv GIZMO_MODULES test;myplug;jpegloader
```

You can also specify a direct loading of a plugin. You can call ***getModule("test")*** to explicit load the plugin test. Remember that a plugin is a good way to add behaviors to your software like new database formats, new image formats, new node types etc.

3.20 Serializing data

The serializing mechanism can be divided into three major entities: data, format and media.

The *data* is the representation of floats, strings, integers and mode structures. Data can be represented by a stream of binary, ascii or other numbers.

The *format* is the logic behind grouping data. If you say that a file always contains two floats and then a string, the format is defined by two floats and a string in that order.

The *media* is the target or source of the data stream. The stream itself doesn't know what it transfers but it knows how to save or load a stream of binary data.

The data is represented by the class ***gzSerializeData*** where you provide methods for writing, reading and pushback of the binary or ASCII representation. The format is defined by grouping several ***gzSerializeData*** classes after each other in a stream and the stream is defined by a ***gzSerializeAdapter*** that has an adapter handle to the stream

3.21 Pipes

A pipe is a two way communication point to point line. When writing to one side of a pipe, the same data can be read from the other end. There is always one pipe end that creates the actual communication and one pipe end that connects to an already created pipe.

You create a pipe communication by creating a *gzPipe* and then do

a pipe->connect(pipeName,TRUE) where pipeName is a string identifying a unique name of the pipe. The ***TRUE*** argument tells the system to create a new pipe. The next ***connect(pipeName)*** will connect to that created pipe and you can use writePipe and readPipe to write and read from the pipe.

4 TUTORIALS

4.1 Messaging and debugging

This tutorial will show you how to print messages and use these to debug your program. Start with creating an empty workspace. Then you need to setup the correct path. In windows this is done by right-clicking on this computer → properties → advanced → environment variables. The path must point to the directory where the GizmoBase dll-file is located.

Then the properties for the project need to be set. The GizmoBase include directories have to be specified as additional include directory. The lib-directories for GizmoBase need to be specified as additional library directory. The last to be done is to specify gzbase.lib as additional dependency.

4.1.1 Hello word!

This example shows how to print the "hello" word using the messaging features in GizmoBase. By using the same technique in your program you will be able to route the output to a graphical GUI or even TCP/IP as your app grows.

GizmoBase uses exceptions whenever there is something strange going on. The errors can be caught by using try and catch statements. To use the file gzExceptions.h needs to be included in your file.

```
#include "gzExceptions.h"

void main()
{
	try
	{

	}

	catch(gzBaseError &error)	// In case of exceptions thrown we want
	{							// to print the message

		error.reportError();	// This is reported through the same
								// message as the above text is
	}
}
```

Then you have to include the file gzDebug.h which is required to use messaging or debugging functions. When creating a message a message level needs to be specified. GizmoBase uses 8 message levels. By default the notice, warning, fatal, assert and always levels are displayed. The NOTICE level can be used to tell the user some kind of normal info, the WARNING level can be used to tell the user that the system works but with some kind of degraded performance. The FATAL level can be used to tell the user that something is really wrong and the app should terminate. A message is created with the line
GZMESSAGE(gzMessageLevel,"The message you want to print");

The message is sent to the default output. If the message level is notice the default output is "GizmoSDK notice::The message you want to print". You will later see how to add your own message manager.

Below follows the code for the complete program which prints a message.

```
#include "gzDebug.h"
#include "gzExceptions.h"

void main()
{
     try
     {

          GZMESSAGE(GZ_MESSAGE_NOTICE,"Hello !");

     }
     catch(gzBaseError &error)
     {

          error.reportError();

     }
}
```

4.1.2 Debug and Trace info

It is possible to change the message level. If you want some debug info you need a lower message level than if you run a release version. To change message level the line ***gzMessage::setMessageLevel(gzMessageLevel)*** is added to your program. Now all messages of this level and all higher levels is displayed.

Change the message level to debug mode and add a debug message in your program.

```
gzMessage::setMessageLevel(GZ_MESSAGE_DEBUG);

GZMESSAGE(GZ_MESSAGE_DEBUG,"Hello !");
```

Now the message “Hello!” is sent to the default output.

Sometimes it is convenient to add trace output in your program to see how the execution flows. When you compile into a release you normally want the trace output to be removed. To do this you start by adding a definition of GZ_DEBUG at the top of your program. This is required to get the trace output when trace commands is used in the code. By removing the GZ_DEBUG define, the trace output is removed.

Now you can add a trace message. This message is sent to the output if the GZ_DEBUG is defined. The complete program follows below:

```
#define GZ_DEBUG

#include "gzDebug.h"
```

```
#include "gzExceptions.h"

void main()
{

     try
     {
          gzMessage::setMessageLevel(GZ_MESSAGE_DEBUG);

          GZMESSAGE(GZ_MESSAGE_DEBUG,"Hello !");

          GZTRACE("Debug info");

          // Trace info about line, code module and
          // information about what version you have
          GZTRACE("Some trace info %s",GZ_DEBUG_INFO("1.0"));
     }
     catch(gzBaseError &error)
     {
          error.reportError();
     }
}
```

The previous code generated the following output.

```
GizmoSDK Debug::Hello !
GizmoSDK Debug::[TRACE] Debug info
GizmoSDK Debug::[TRACE] Some trace info File:C:\projekt\amo\Dev\gizmo\main\main.cpp  Line:22  Ver:1.0
```

It is easy to remove the trace messages, just by comment the GZ_DEBUG in the top of the program. The GZMESSAGE will always be generated if the correct message level is active.

4.2 Memory debugging

You can use a built in memory debugger. By defining GZ_MEM_DEBUG at the top of your program or for your whole project you can catch memory leaks. On the WIN32 model the output is routed to the trace console (ATLTRACE) in Visual Studio or on external trace consoles. On UNIX it is written to stdout.

```
#define GZ_MEM_DEBUG

#include "gzMemory.h"
#include "gzExceptions.h"    // Exception management.

void main()
{
     try
     {
          // To show mem debug
          gzMessage::setMessageLevel(GZ_MESSAGE_MEM_DEBUG);

          gzInt *a=new gzInt;
     }

     catch(gzBaseError &error)
     {
          error.reportError();
     }
}
```

The following output is generated..

```
--!! Lost mem 4 bytes at 0x110ef844 index 0 ->
File:C:\projekt\amo\Dev\gizmo\main\main.cpp  Line:11  Ver:1.1 Beta 2
```

4.3 Referenced memory

If you are used to working with normal pointers and memory allocations, you now how easy it is to forget to deallocate memory or how hard it is when you want to share memory, and how to decide who is responsible to free the shared memory.

The answer is referenced memory. It is basically a reference counter that is incremented for each user that wants the data and decremented as each user releases its interest in the data. When the last user releases its handle, the memory is deleted.

```
#define GZ_MEM_DEBUG

#include "gzReference.h"
#include "gzExceptions.h"    // Exception management.

class MyRefClass : public gzReference // Inheritance from refrence base
{
public:

};

void main()
{
     try
     {
          // To show mem debug
          gzMessage::setMessageLevel(GZ_MESSAGE_MEM_DEBUG);

          MyRefClass *pMyRefClass = new MyRefClass;

          pMyRefClass->ref();          // Increment reference
          pMyRefClass->ref();          // Increment reference

          pMyRefClass->unref();   // Decrement reference
          pMyRefClass->unref();   // Deleted

          // The instance is deleted as the last referee
          // makes his unref

     }
     catch(gzBaseError &error)
     {
          error.reportError();
     }
}
```

No reported memory leaks. Note that the GZ_MEM_DEBUG is defined !!

4.4 Smart Pointers

A smart pointer is actually an instance that increments a reference to *gzReference* memory and decrements upon destruction of its own instance. This could be used as automatic garbage collection. The best way to use them and still maintain readability so the reader is not confused with normal pointers is to use the get() and set() methods. You can use them as normal pointers because they have the proper operators defined, but this might confuse a reader of your code to exchange them with normal pointers.

```
#define GZ_MEM_DEBUG

#include "gzReference.h"
#include "gzExceptions.h"    // Exception management.

class MyRefClass : public gzReference
{
public:

     int a;
};

void main()
{
     try
     {
          // To show mem debug

          gzMessage::setMessageLevel(GZ_MESSAGE_MEM_DEBUG);

          // The smart pointer is a template class.
          gzRefPointer<MyRefClass> myRefClassPtr = new MyRefClass;

          myRefClassPtr->a=10; // assign the member var a = 10

          int b=myRefClassPtr->a; // let b = member var a

          // or perhaps better readability
          // that shows that they are not usual pointers

          myRefClassPtr.get()->a=20; // assign member var a=20

     }
     catch(gzBaseError &error)
     {
          error.reportError();
     }
}
```

Note that there are NO memory leaks, because when the variable myRefClassPtr goes out of scope, the ref count is decreased and this last unref deletes the instance of the MyRefClass data. This is hidden from the user and is automatically handled by the system.

4.5 Working with threads

Working with threads is very easy in GizmoBase. A thread allows parallel execution of software. The example below shows how the main thread starts another thread and then sleeps 5 seconds. The other thread prints some info and then sleeps a second until it is stopped.

```
#include "gzThread.h"
#include "gzExceptions.h"    // Exception management.
#include <iostream.h>

class MyThreadClass : public gzThread
{
public:

     gzVoid process()    // this is the magic name of the process
     {
          while( ! isStopping() ) // run until we are stopped
          {
               cout<<"helloooo"<<endl;
               gzSleep(GZ_SLEEP_SECOND);    // sleeeeeeep
          }
     }
};

void main()
{
     try
     {
          MyThreadClass thread;

          thread.run(); // start the thread

          gzSleep(GZ_SLEEP_SECOND * 5);     // sleep 5 seconds

          thread.stop();// and stop the thread

     }
     catch(gzBaseError &error)
     {
          error.reportError();
     }
}
```

4.6 Working with pipes

Working with pipes is an easy way to create communication between threads and processes. A pipe is a kind of point to point communication and the following examples shows how to communication from one pipe to other listening pipe clients. In this case we got two clients that are serializers. The first one sends a string “test” onto the pipe and all other listening clients will receive that data.

```
    // Create one pipe server that multiplexes data between pipes
    // named "gizmo3d"
    gzPipeServer server("gizmo3d");

    server.runServer();

    gzSleep(1000);// Wait for pipe creation

    // Create two serializers

    gzSerializeAdapter *adapter1,*adapter2;
    adapter1 = gzSerializeAdapter::getURLAdapter("pipe:gizmo3d");
    adapter2 = gzSerializeAdapter::getURLAdapter("pipe:gizmo3d");

    if(adapter1 && adapter2)
    {
        // write a string to pipe
        gzSerializeString a="test";

        a.write(adapter1);

        // wait for data
        while(!adapter2->hasData())
            gzSleep(0);

        // read data
        gzSerializeString b;

        b.read(adapter2);

        GZMESSAGE(GZ_MESSAGE_NOTICE,"Got data '%s' from pipe",(const char
*)(gzString)b);

        delete adapter1;

        delete adapter2;

    }

    // terminate and wait for all pipes to shut down
    server.terminateServer();
```

5 EXAMPLES

5.1 Using a pointer list

```
gzList<gzULong> theList;

gzULong a=10;
gzULong b=99;

theList.insert(&a);     // Insert address pointers
theList.insert(&b);

gzULong *item;// A temporary pointer
gzListIterator<gzULong> iterator(theList);

while(item=iterator())
{
     cout<<*item<<endl;
}
```

5.2 Using a reference pointer list

```
gzRefList< gzRefData< gzBaseClass<gzULong> > > theList;


theList.insert(new gzRefData< gzBaseClass<gzULong> >(10) );      // Insert
address pointers
theList.insert(new gzRefData< gzBaseClass<gzULong> >(99));

gzRefData< gzBaseClass<gzULong> > *item;   // A temporary pointer
gzListIterator< gzRefData< gzBaseClass<gzULong> > > iterator(theList);

while(item=iterator())
{
     cout<<(gzULong)*item<<endl;
}
```

NOTE !!! This example automatically frees the referenced data. All classes derived from the *gzReference* class can be used with this type of lists and dictionaries. In the example above we can also see how to add reference management to a class that is not derived from the *gzReference* class, by using the *gzRefData* in combination with the *gzBaseClass* template. Classes (not primitives like float etc) can be used without the *gzBaseClass* template

5.3 Message Receiver

```
#include "gzDebug.h"
#include "gzExceptions.h"
#include "iostream.h"

#ifdef WIN32
#include "atlbase.h"
#endif

class MyMessageReceiver : public gzMessageReceiverInterface
```

```
{
public:

     MyMessageReceiver()
     {
          gzMessage::addMessageReceiver(this);
     }

     virtual ~MyMessageReceiver()
     {
          gzMessage::removeMessageReceiver(this);
     }

     gzVoid onMessage(const gzString& sender , gzMessageLevel level , const
char *message)
     {
          #ifdef WIN32
               ATLTRACE("%s from %s\n",message,(const char *)sender);
          #else
               cout<<message<<endl;
          #endif
     }

};


void main(int argc , char *argv[] )
{
     try       // To catch all GizmoBase exceptions
     {
          gzMessage::setMessageLevel(GZ_MESSAGE_DEBUG);

          MyMessageReceiver theReceiver;

          GZMESSAGE(GZ_MESSAGE_DEBUG,"Hejsan");
               // Outputs a debug level message

          // Outputs a TRACE message, remove when GZ_DEBUG is undefined
          GZTRACE("Debug");
     }
     catch(gzBaseError &error)
          // In case of exceptions thrown we want to print the message
     {
          error.reportError();
     }
}
```

5.4 Dynamic Plugin

```
#include "gzModule.h"
#include "stdio.h"
#define VERSIONSTR "MODULE: plugin ver 1.0"

class MinModul : public gzModule
{
public:
     GZ_DECLARE_TYPE_INTERFACE;   // Dynamic RTTI info
```

```
    gzVoid onEvent( gzModuleEvent event)
    {
        switch(event)
        {
            case GZ_MODULE_ADD:
                GZMESSAGE(GZ_MESSAGE_DEBUG,"Loading Plugin Module");
                break;

            case GZ_MODULE_REMOVE:
                GZMESSAGE(GZ_MESSAGE_DEBUG,"Unloading Plugin Module");
                break;
        }
    };

    gzDynamicType   version(GZ_DYNAMIC_ATTRIBUTE_LIST)
    {
        GZMESSAGE(GZ_MESSAGE_DEBUG,(const char *)VERSIONSTR);

        return gzDynamicTypeVoid();
    }

    //----------- Dynamic object methods --------------------------------

    virtual gzDynamicType   invokeMethod(gzULongLong
IID_method,GZ_DYNAMIC_ATTRIBUTE_LIST)
    {
        switch(IID_method)
        {
            case IID_VERSION:
                return version(GZ_DYNAMIC_ATTRIBUTE_LIST_IMP_FORWARD);

        }
        return gzDynamicTypeError(GZ_DYNAMIC_ERROR_NOT_IMPLEMENTED);
    }

    virtual gzBool supportMethod(gzULongLong IID_method)
    {
        switch(IID_method)
        {
            case IID_VERSION:
                return TRUE;
        }
        return FALSE;
    }

    virtual gzULongLong getMethodIID(const gzString &method)
    {
        if(method==IIDS_VERSION)
            return IID_VERSION;

        return 0;
    }

    virtual gzArray<gzDynamicMethodID> queryAllMethodIID()
    {
        gzArray<gzDynamicMethodID> array;

        array.setSize(1);

        array[0].IID_method=IID_VERSION;
```

```
          array[0].IIDS_method=IIDS_VERSION;

          return array;

     }
     //----------- Clone interface ---------------------------------------

     gzReference *clone() const
     {
          return (gzReference *)new MinModul(*this);
     }

};

GZ_DECLARE_TYPE_CHILD(gzModule,MinModul,"MyModule");

GZ_DECLARE_MODULE(MinModul);
```

The example above shows a plugin that supports a generic method named “version”. You can add your own methods, Just make sure the last three methods are extended with you method invocation information. Parameter passing is made by the *gzDynamicType* that is able to pass any kind of parameter.

Most plugins just listens to the onEvent method which is called when the plugin is loaded/unloaded, to register notifiers or loaders, behaviors, etc.

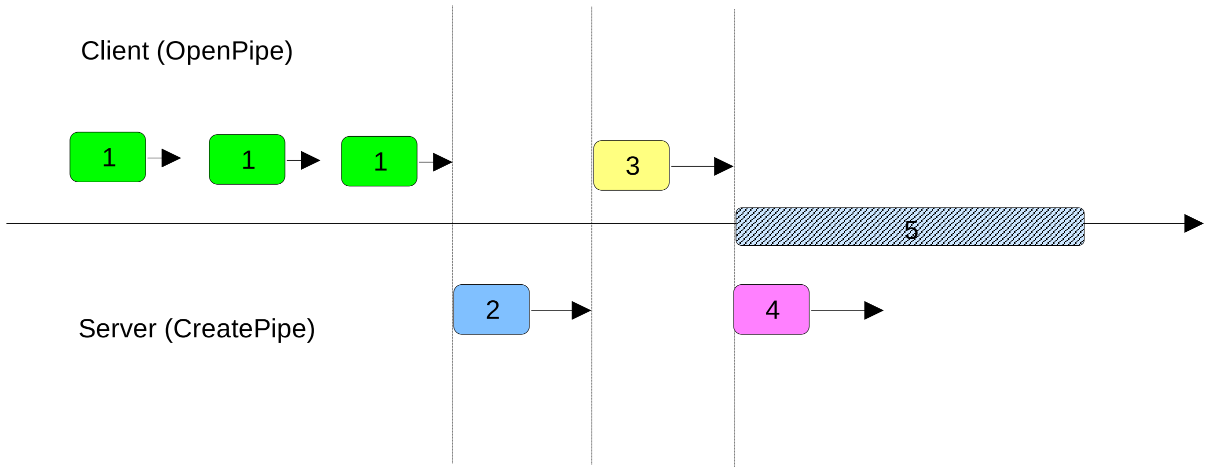
6 FAQ

7 APPENDIX

7.1 gzPipe Communication Schema

The gzPipe adapter uses a communication schema when establishing the tcp communication using a broadcast request/acknowledge technique.

The following picture outlines the communication between the client that starts to request the connection and the server that responds.



In the picture above we have started a server that is listening to broadcast or multicast.

The client starts by issuing ***open requests*** (1) until he gets an answer from the server ***ack*** (2). The clients responds to this answer by issuing his own answer ***ack*** (3). Finally the server ends the chat with an ***open ok*** answer (4) and starts the tcp communication (5)

The **client** sends his requests by default on ***port 8100***.
The **server** responds to clients by default on ***port 8101***.

7.2 gzPipe Communication packages

All data below is written with big endian order.

ULong	is 4 bytes unsigned long	0..0xFFFFFFFF
UShort	is 2 bytes unsigned short	0..0xFFFF
UByte	is 1 byte unsigned byte	0..0xFF
String	is variable count of non zero bytes and terminated with a zero byte	
Guid	is the same as String but with unique contetnts according to the GUID standard (Can use normal text if required)	

All packages are started with a **MAGIC** identifier to allow some level of garbage on the line

7.2.1 Generic Package Header

ULong	MAGIC	`0xab36f3e8`

UShort	Length	The length of the trailing package in bytes
--	--	--

7.2.2 `GZ_PIPE_COMMAND_OPEN (1)`

ULong	MAGIC	`0xab36f3e8`
UShort	Length	The length of the trailing package in bytes
UByte	0	Command OPEN from client
String	##	Pipe Name
Guid	{…}	Client Pipe ID
String	##	Option String from client
ULong	1..	Request ID (Increased for each new session)

7.2.3 `GZ_PIPE_COMMAND_ACK_SERVER (2)`

ULong	MAGIC	`0xab36f3e8`
UShort	Length	The length of the trailing package in bytes
UByte	1	Command ACK from server
String	##	Pipe Name
Guid	{…}	Client Pipe ID that issued the OPEN request
Guid	{…}	Server Pipe ID
String	##	Option String from server

7.2.4 `GZ_PIPE_COMMAND_ACK_CLIENT (3)`

ULong	MAGIC	`0xab36f3e8`
UShort	Length	The length of the trailing package in bytes
UByte	2	Command ACK from client
String	##	Pipe Name
Guid	{…}	Server Pipe ID that sent us ACK
Guid	{…}	Client Pipe ID

7.2.5 `GZ_PIPE_COMMAND_OPEN_OK (4)`

ULong	MAGIC	`0xab36f3e8`
UShort	Length	The length of the trailing package in bytes
UByte	3	Command OPEN OK from server
String	##	Pipe Name
Guid	{…}	Client Pipe ID that issued the ACK
Guid	{…}	Server Pipe ID
UShort	8001..	Current available port number for tcp

7.3 Defined Types

```
gzBool;         //!< Boolean type
gzBitfield;     //!< 16 bits bitfield
gzByte;         //!< Signed byte value -128 to +127
gzChar;         //!< Signed byte char -128 to +127
gzUByte;        //!< Unsigned byte value 0 to 255
gzShort;        //!< Signed 16 bit integer
gzUShort;       //!< Unsigned 16 bit integer
gzInt;          //!< Signed 32 bit integer
gzUInt;         //!< Unsigned 32 bit integer
gzLong;         //!< Signed 32 bit integer
gzULong;        //!< Unsigned 32 bit integer
gzFloat;        //!< IEEE 754 32 bit float
gzDouble;       //!< IEEE 754 64 bit double
gzHandle;       //!< Unsigned 32 bit handle
gzWideChar;     //!< Wide character value   (wchar_t)
gzLongLong;     //!< 64 bit signed integer
gzULongLong;    //!< 64 bit unsigned integer
gzVoid;         //!< Void ( no result )
gzMemPointer;   //!< Void * ( anything )
gzMemSize;      //!< Size of memory chunk   (gzULongLong)
gzMemOffset;    //!< +- offset of memory chunk   (gzLongLong)
```

7.4 Defined Constants

```
//! Definition of PI etc.
const gzDouble     GZ_PI=3.14159265358979323846264338;
const gzDouble     GZ_2PI=GZ_PI*2;
const gzDouble     GZ_PI_HALF=GZ_PI/2;
const gzDouble     GZ_PI_QUARTER=GZ_PI/4;

const gzDouble     GZ_INV_PI=1.0/GZ_PI;
const gzDouble     GZ_INV_2PI=1.0/GZ_2PI;
const gzDouble     GZ_INV_PI_HALF=1.0/GZ_PI_HALF;
const gzDouble     GZ_INV_PI_QUARTER=1.0/GZ_PI_QUARTER;

const gzFloat      GZ_INV_255=1.0f/255.0f;

const gzFloat      GZ_FLOAT_ONE=1.0f;
const gzFloat      GZ_FLOAT_ZERO=0.0f;

const gzDouble     GZ_DOUBLE_ONE=1.0;
const gzDouble     GZ_DOUBLE_ZERO=0.0;

const gzReal       GZ_REAL_ONE=1.0f;
const gzReal       GZ_REAL_ZERO=0.0f;

const gzDouble     GZ_DEG2RAD=0.01745329251994329576923690766667;
const gzDouble     GZ_MILS2RAD=9.8174770424681038701957605625011e-4;
const gzDouble     GZ_RAD2DEG=57.295779513082320876798154873916;
const gzDouble     GZ_RAD2MILS=1018.5916357881301489208560866473;

const gzDouble     GZ_SQRT_2=1.4142135623730950488016887242097;
const gzDouble     GZ_SQRT_3=1.7320508075688772935274463415059;
```

8 LINKS

Here are some recommended links that you can take a look at

http://www.gizmosdk.com	The GizmoSDK web site