OpenSceneGraph meta-data system

Specifications proposal

Describes a proposal for a meta-data system integrated in the core OSG library.

Revision date	Authors
2011-04-15	Gregoire Tarizzo, Sukender (Benoit Neil)

Table of contents

1. Foreword

Some OSG users felt the need of adding meta-data into core OSG[1]. Unfortunately, they have several incompatible uses and ideas on how to deal with this. The goal of this document is to ensure OSG community mostly agrees on a common idea of what the OSG meta-data system should be, before implementation.

Please send your comments and ideas to the osg-users mailing-list or directly to the original authors, which will be in charge of the first implementation of this meta-data system.

So, what is a meta ? On this definition, OSG community already seem to agree a meta should be a **“name-value” pair** (where “name” is a character string). The system described here uses this definition as a start point.

2. Purpose

As Peter Amstutz told, the goal is "[to] add a baseline metadata system as standard (as opposed to the current do-what-ever-you-want situation with UserData)".

Meta-data system is thus meant to:

- Add custom user data
- Somehow relate this data to the graph
- Allow this in a more flexible way than the current implementation
- Add the ability for readerwriters to "return" values (as the Collada plugin currently does with an ugly const_cast<>)

2.1. Constraints

The following constraints have been kept in mind when building these specifications:

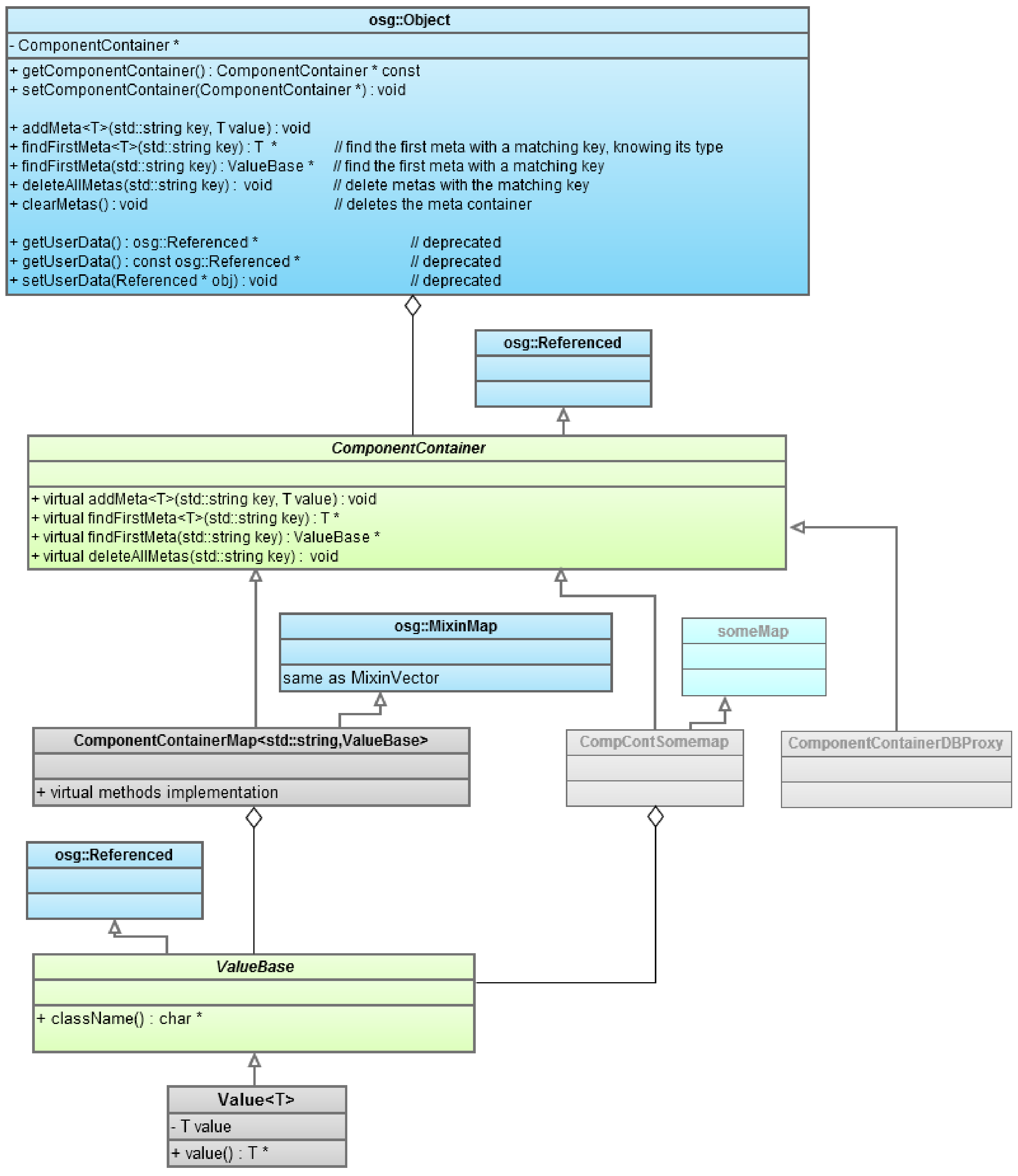
- **No noticeable overhead** (memory, CPU, or development time), especially regarding to the initial goal of a scene graph
- Meta must be **typed** values
- Metadata system must be **extensible**, to:
    - Store in-graph or out-of-the-graph data, as user wishes
    - Let the user choose allowing or forbidding duplicate keys
    - Be as general-purpose as possible
- Default choices must exist for an almost-zero effort implementation of basic cases
- Meta serialization by OSG reader/writers must be straightforward to implement
- System should supersede the existing “_userData” or “descriptions” systems, with something more powerful, and without loss of functionality.
- System should be as backward-compatible as possible

1 OSG forum links:
http://forum.openscenegraph.org/viewtopic.php?t=7655
http://forum.openscenegraph.org/viewtopic.php?t=3391 then http://forum.openscenegraph.org/viewtopic.php?t=3961
http://forum.openscenegraph.org/viewtopic.php?t=4285

3. Specifications

This section describes the choices made for the meta-data system, and eventually the reasons why some other choices have been discarded.

OSG MetaData prototype


osg::Object
- ComponentContainer *
+ getComponentContainer() : ComponentContainer * const
+ setComponentContainer(ComponentContainer *) : void
+ addMeta<T>(std::string key, T value) : void
+ findFirstMeta<T>(std::string key) : T * // find the first meta with a matching key, knowing its type
+ findFirstMeta(std::string key) : ValueBase * // find the first meta with a matching key
+ deleteAllMetas(std::string key) : void // delete metas with the matching key
+ clearMetas() : void // deletes the meta container
+ getUserData() : osg::Referenced * // deprecated
+ getUserData() : const osg::Referenced * // deprecated
+ setUserData(Referenced * obj) : void // deprecated
osg::Referenced
ComponentContainer
+ virtual addMeta<T>(std::string key, T value) : void
+ virtual findFirstMeta<T>(std::string key) : T *
+ virtual findFirstMeta(std::string key) : ValueBase *
+ virtual deleteAllMetas(std::string key) : void
osg::MixinMap
same as MixinVector
someMap
ComponentContainerMap<std::string,ValueBase>
+ virtual methods implementation
CompContSomemap
ComponentContainerDBProxy
osg::Referenced
ValueBase
+ className() : char *
Value<T>
- T value
+ value() : T *

3.1. Containers

- osg::Object has a pointer to a meta container (in place of the current _userData), here named "ComponentContainer":
    - Container is thus optional. This way the user is free to use this functionality or not (ie. an instance of the container is created only if a meta is used). This is one reason why osg::Object **is not** a container.
    - Container may be owned by an Object, or shared with other Objects, or be owned by complex out-of-the-graph systems.
- Containers have a common abstract base class
    - This allows users to have the container of their choice. This is another reason why osg::Object is not a container. For instance, we could have classes inheriting from:
        - std::map (no duplicates)
        - std::multimap (allows duplicates)
        - std::vector<std::pair<> > or fixed-size arrays (less small memory allocations)
        - std::unordered_map (C++0x) / boost:: unordered_map (no duplicates, fast look-up of strings using a hash value)
        - Any user-defined container, including a relational **database** management system (or a proxy to such a system). This would allow very complex frameworks to be plugged in the meta-data system.
    - The overhead caused by virtual calls is justified by the flexibility it provides. Indeed, users of meta seem to have very different purposes (especially regarding to the handling of duplicates and the amount of meta-data managed).
- Containers have pointers to meta values
    - To allow mixed-types values in a single container
    - To avoid copying data each time we push or access a value. Indeed, accessing strings may be frequent and inefficient if copied on each read.

3.1.1. Default container

What we need here is a map container as the default, for several reasons:

- In basic cases, users don't need duplicate values
- std::vector<std::pair<> > may not be intuitive as a map (and it allows duplicates)
- unordered_map isn't a possible choice as OSG users may not all use a C++0x compiler, or have boost libraries as a dependency.

But as "osg::MixinVector", the problem with the std::map container is that it does not have a virtual destructor (which can be a problem when destroying a class that inherits the map class). So we propose something similar to the osg::MixinVector and will implement[2] the osg::MixinMap for meta-data storage.

The standard container for meta-data will thus be "ComponentContainerMap", which inherits from ComponentContainer and MixinMap. With this, we will have a simple but effective way to handle single key typed meta-data.

3.1.2. Thread-safety

The fact that the default container is not thread-safe doesn't mean all the containers can't be. Nothing forbids the user from using a "ComponentContainerMutexedMap" or whatever.

2 Well, actually it is already implemented :)

We feel that thread-safe containers should not be the default, as it may slow down processes too much. However, thread-safe containers may be proposed somewhere else in OpenSceneGraph.

3.2. Convenience

One important point of the meta-data system is to make it very easy to use for basic uses.

- Container base class has minimal virtual methods:
  - findFirstMeta(): Find the first meta-value for a given key, or NULL.
    - For no-duplicates containers, it's the standard “find()” method
    - Else it returns the first value having such a key – hence the method name.
  - findFirstMeta<T>() : as findFirstMeta() with casting of the result.
    - static_cast<>() has been considered, but found too dangerous for a convenience method.
    - A type test prior to a static_cast<>() (ie. “`if (T::staticClassName() == value.className())...`”) has been considered too, but leads to unsolved problems when querying a base type of the actual meta-value.
    - dynamic_cast<>() has thus been preferred. Convenience free functions using static_cast<> may be coded elsewhere, or on the user's side.
  - deleteAllMetas(): Delete all the values designed by a specific key.
  - clearMetas(): Remove all metas and delete container.
- osg::Object has convenience non-virtual methods for a shorter writing of basic cases (ie. adding and removing a meta to an object).
  - When adding a meta, a default container is created if it wasn't done before. If the user wants a particular container, (s)he has to create it and add it to the object before adding any meta-data.
  - Also when the meta container is emptied, it's automatically unreferenced, since empty containers are useful only in very specific cases.

3.3. Meta-data values

Most containers store ValueBase pointers. In this subsection, we deal with such containers (= not specific ones such as proxies to a database).

Values have to be typed...

- That's the reason for a ValueBase interface, providing the tools to achieve that. Moreover, ValueBase allows the containers to have pointers to mixed-types, implying such a base class.
- Initial design of metas intended to have basic types as values (int, float, string...), or at least non-virtual classes to handle values (= ValueBase having no virtual calls). However some needs made these classes derive from osg::Referenced and thus be virtual classes:
  - Containers would have difficulties to handle mixed-types values. A “void*” container has been considered as way too dangerous and excluded from the very beginning of the current study.
  - Users may need to read values whose type is unknown at a given point in the program. Values have to know their own type then.
    - “typeid” is not reliable when serializing/deserializing values with different executables.
  - Metas may be shared, especially for values with large amount of data.
- Values may also be based on boost::any, for users having boost libraries as a dependency.

The value discussion was a tough point. We wanted a generic container, able to store typed values. In order to do that, we had to keep somewhere the type of the value that was stored. For obvious reasons, ValueBase had to inherit from Referenced.

As ValueBase would inherit from Referenced, there would have no way to keep it non-virtual. Hence we decided to implement a macro for virtual calls of the ValueBase methods (like it's done with the META_Node macro).

At some point we had to use dynamic_cast when getting a value. At first we thought of a solution with a basic type comparison, but it is not appropriate when retrieving a “Pie” inheriting from “Cake” as a “Cake *”.

Another point to discuss is the need (or not) for clone()/cloneType() methods on Values, as they exist on Node. Maybe this would add too much data (two pointers) for small meta (such as int)?

3.3.1. Thread-safety

Here thread safety entirely depends on the type of the value. `Value<int>` is typically not thread-safe, whereas `Value<MySafeStructure>` is.

4. Integration and backward-compatibility

4.1. Existing code

It was our wish to replace the “userData” and “description”, while being backward compatible. So we have to make the following changes:

- Deprecation of getters and setters for “userData” and “description”. These now call the meta-data system.
  - Two metas names will be reserved for _userData and _descriptionList: “`$OSG_USERDATA_META$`” and “`$OSG_DESCRIPTIONS_META$`”.
  - Types of replacement metas are of course “`osg::Referenced`” and “`std::vector<std::string>`”, respectively.
- Deletion of the _userData and _descriptionList fields
- Modification of a few classes that used the _userData field directly. The fact that these attributes are protected and not private creates a risk, where sub-classes call the attribute without going through the getter or setter.

Regarding to backward-compatibility constraint, multiple choices have been studied. We (initial authors of this document) feel that the removal of “_userData” (as well as “description”) is an acceptable drawback.

We feel the modifications made to OSG by this last point are really minor, as just a few calls have to be replaced by the getter or the setter. Users' code will be of course very straightforward to change. We guess that few users directly access userData without using getter/setter.

Other choices we studied also had drawbacks. Noticeably the fact that having a meta-system alongside “userData” and “description” would burden osg::Object with more variables, duplicate features, and let the risk of a divergence of usages (and thus users' code). We feel that unifying the system is far more important than having 99% compatibility instead of 100%.

4.2. Performance

The modifications done to OSG do not have any noticeable negative impact on performance, while keeping backward compatibility and doing something quite clean and flexible. Among other things:

- osg::Object class keeps the same size in memory with this meta-data system.
  - userData has been replaced with componentContainer
  - No virtual calls added
- Accessing “userData” or “description” as it is done until now only adds the cost of one or two string comparisons (depending on if you use one or both).
- osg::Node is smaller since it has been cleared from “descriptions”, moving the information in the metas. Nodes having no description will thus be a few bytes lighter[3].

3 If I (Sukender) understood well, std::vector<> size ranges from 12 bytes on a 32 bits system to 48 bytes on a 64 bits system using some compilers (namely MSVC) in debug mode with iterator debugging.

4.3. ReaderWriters & serialization

The idea behind (de)serialization of metas has clearly to be studied further. We guess that common types (int, double, string...) should be handled by current ReaderWriters. But custom types should also be serializable, maybe using the extensible serialization framework. As well, deserialization may need a factory to allocate values from a class name. The help from a serialization specialist would be greatly appreciated here!

As a fallback, Values should be serialized as userData is in current implementations.

4.3.1. Return values for readers

Readers now cannot return meta informations easily. For instance, the Collada plugin uses the osgDB::Option structure to pass values to the caller (such as the measure unit). But this structure is const and has not been designed for such a role.

We thought at first than an additional (non-const) parameter should be passed in order to let readers write meta data. But this involves a bit of changes and forces user's code to somehow get that structure.

What should be done by readers then is to simply write meta-data attached to the root node of the returned graph. This has the drawback of not being able to return data alongside an empty graph or if reading failed. However this may cover most needs and be very straightforward to implement and use.

4.4. Integration with databases

This document does not intend to cover the case of integration with databases systems, but clearly the aim is to allow it if user needs it. For example, users may define, say, “DatabaseProxy”, which inherits from “ComponentContainer”. This “DatabaseProxy” would access a database to read and write meta values.

In order to associate nodes and values in the database, there should be a way to uniquely identify an object (or node). Among others:

- Memory address: it is straightforward but does not support serialization.
- Node name (or other meta-information): it isn't safe enough to ensure uniqueness, unless user does.
- Identification via a numbered node path:
  - This has been implemented in osgWorks by Chris 'Xenon' Hanson (and in several other private or public codes). For example, “0 1” is the second child under first child of the root.
  - This requires to not modify the scene graph, or consider a subgraph instead of the original graph.
  - This does not work for lone Objects (= classes not derived from Node). However this should not be an issue for most applications.

Regarding these considerations, the choice of object/node identification should be left to the user.

One point which could be discussed is the presence of back-references from the Value to its parent Containers (vector of pointers maybe), as well as from the Container to its parent Objects. These backward references may help systems not using node identification as described above.

4.5. Ideas and future work

- Default container should be changeable by the user. We may think about:
  - A callback to create the default container. This thus should be in a singleton to avoid each osg::Object being burdened by an additional pointer.
  - A prototype container, which has a clone() method.
- The node name (getName() / setName()) could even be moved in the meta-data system, making the class even lighter.
- Providing some different containers may be a good idea:
  - Thread-safe map with virtual destructor
  - Unordered map equivalent
  - etc...
- Make value accept ValueVisitors, a bit as designed in boost.any library. This would thus be easy to apply operations on values of unknown types.

5. Code samples & Examples

5.1. Implementing Value<T>

As in osg::Node, here is an useful define for our Value<T> classes.

```
#define META_Value(type) \
        virtual const char * className() const { return "Value_" #type; } \
        virtual type * value() { return _value; } \
```

Should these methods be included too?

```
        virtual osg::Object * cloneType() const { return new type(); } \
        virtual osg::Object * clone(const osg::CopyOp& copyop) const { return new type (*this,copyop); } \
```

For reference, here is the original Node code:

```
#define META_Node(library,name) \
    virtual osg::Object* cloneType() const { return new name (); } \
    virtual osg::Object* clone(const osg::CopyOp& copyop) const { return new name (*this,copyop); } \
    virtual bool isSameKindAs(const osg::Object* obj) const { return dynamic_cast<const name *>(obj)!=NULL; } \
    virtual const char* className() const { return #name; } \
    virtual const char* libraryName() const { return #library; } \
    virtual void accept(osg::NodeVisitor& nv) { if (nv.validNodeMask(*this)) { nv.pushOntoNodePath(this);
        nv.apply(*this); nv.popFromNodePath(); } } \
```

5.2. Basic cases

5.2.1. Store a value

```
osg::Object * o = new osg::Object;
o->addMeta<int>("cakeNumber", 4);
o->addMeta<myObj>("cakeRecipe", someObject);
```

5.2.2. Retrieve a value, knowing its type

```
int * cakeNumber = o->findFirstMeta<int>("cakeNumber");
if (cakeNumber) some_function(cakeNumber);
```

5.2.3. Retrieve a value (unknown type)

```
ValueBase * cakeNumber = o->findFirstMeta("cakeNumber");
if (!cakeNumber) return;
std::cout << "Cake number type: " << cakeNumber->className() << std::endl;
if ( (Value<int>* intVal = dynamic_cast<Value<int>*>(cakeNumber)) != NULL)
{
    std::cout<<"Cake number: " << intVal->getValue() << std::endl;
}
else //...


// Or using className(), if you wish the exact type
```

```
ValueBase * cakeNumber = o->findFirstMeta(“cakeNumber”);
if (!cakeNumber) return;
if (cakeNumber->className() == "Value_int") // ...
```

5.2.4. Utilities

```
// We also may provide utilities, such as:

/// Equivalent of osg::Object::findFirstMeta() which throws if meta doesn't
exist or is of inappropriate type.
template<typename T>
T & findFirstMeta(osg::Object * o, const std::string & name)
{
	T * v = o->findFirstMeta<T>(name);
	if (!v) throw std::runtime_error("Could not get meta");
	return *v;
}
```