Delta3D Developer's Style Guide

For Delta3D Developers

v1.1.1

Formatting and Syntax

- No Hungarian notation required.
- Add a newline at the end of every file.
- Indentation - 3 spaces, no hard-tabs
- Case
    - Namespaces - camelback notation, starting with "dt" (e.g. `namespace dtCore`)
    - Classes - camelback notation, starts with uppercase (e.g. `MySwellClass`)
    - Filenames - all lower case (*.cpp and *.h). (e.g. `myfile.cpp`)
    - Directories - match the case of the namespaces (e.g. `src/dtChar`)
    - Enums/Macros - all uppercase separated by underscores (e.g. `#define YOU_ARE_SPECIAL`), but please avoid if possible by using `const`, `inline`, and templates instead.
- Class Members
    - Internal variables - names begins with lower case 'm' (e.g. `int mNiceNumber`)
    - Methods - camelback, starts with uppercase letter (e.g. `void CalcNumber()`)
- Braces - BSD (in column) style. Example:

```
if(true)
{
}
```

- All loops and conditional statements should have their enclosed statements within a brace pair, even if there is only one statement enclosed.
- Comments - use JavaDoc/Doxygen format. Full comment in headers describing parameters, return values, etc. Implementation files should preceed each method with a row of 80 slashes (e.g., ///////...).
- Limit line length to 120 columns wide; try to stay with 80 columns if practical.
- `#include` should use the brackets instead of quotes unless the class must include a file from the local directory. For example, `#include <dtCore/camera.h>`.
- First `#include` in an implementation file should be the class' header file.
- Do not place a semicolon after a namespace declaration.
- Pointers, references - "*" and "&" should be attached to the variable type and not the variable name (e.g. `"int* Foo(float* myFloat)", "Object* myObj")`.
- Parameter spacing - separate parameters with spaces with no spaces next to parens (e.g., `"void Foo(float* myFloat, int myInt);")`
- Method spacing - no spaces are added between the method name and the parenthesis (e.g., "`int x = GetValue(12);`")
- Methods should be listed in the implementation file to roughly match the order in the header file.
- Class member initializations should be preceded by a comma and space. Example:

```
MyClass()
: mMemberA()
, mMemberB()
, mMemberC()
{
   ...
}
```

Coding Practices

- Force the compiler to check for redundant includes with `#ifndef`, `#define`, `#endif`
- Use "Set" and "Get" for class members (e.g. `SetName()`, `GetName()`)
- No C-style casts. Use `dynamic_cast`, `static_cast`, or `reinterpret_cast`. If you are casting a primitive, try to use a C++ style cast like `float(x)` rather than `(float)x`. The exception to this is that g++ has a bug with two named types, so you have to do `(unsigned long)(x)`.
- No `#pragma` directives (they are compiler dependant).
- Singleton classes use `GetInstance()` which return by reference.
- Reference-counted classes (i.e. those that have `osg::Referenced` as a base) must have protected destructors. Store instances of Delta3D reference-counted classes in `dtCore::RefPtr`, but be careful to avoid circular references. Use an `dtCore::ObserverPtr` in circular reference cases.
- Don't use a `char*` for a string, instead use a `std::string`. When passing `std::string` to a function, use `const str::string&` unless you plan to change the string. When returning a string that is a member of a class, use `const std::string&` whenever possible. If you are returning a temporary, return `const std::string`.
- For `std::string` that are static for their lifetime, consider using the `RefString` instead.
- Prefer using references instead of pointers. If `NULL` is in the valid domain for your parameter or return value, the pointers are fine.
- Prefer using STL containers and algoritms over custom solutions.
- Don’t make methods that take RefPtrs as arguments or return them. The exception to this is when a method is creating something for you and returning it. In that case, returning the RefPtr tells the caller that they have the only handle to the object.
- Don’t use `throw` clauses on function declarations.
- Use Forward Declarations when possible. Do not use a Forward Declaration for variables contained in a RefPtr; `#include` the header file instead.
- Use the `const` keyword whenever possible: const methods, const parameters, const return values.
- Avoid long methods: Methods should be short and perform one function. If a long method is performing multiple functions in sequence (typically separated by comments), consider extracting the functionality into a separate method.
- Avoid long message chains: If a class asks one object for another object, which then asks for another object, consider an alternative solution to break the dependency list. See how the final object is used. Perhaps it can be supplied to the class directly, or obtained a different way.

*** from the mycoolclass.h file ***

```
#ifndef MYCOOLCLASS_INCLUDED
#define MYCOOLCLASS_INCLUDED

#include <dtCore/base.h>

namespace dtCore
{
   class MyOtherClass; ///<forward declaration

   /**
    * This is a long description and includes what this class is used for,
    * how to use it, and references to related classes.
    */
   class MyCoolClass : public dtCore::Base
   {
   public:
      MyCoolClass();

      /**
       * Set the name of this class.
       * @param name : the new name
       * @param overwrite : true to replace existing name
       */
      void SetName(const std::string& name, bool overwrite);

      /**
       * Get the name of this class.
       * @return The class's name
       */
      const std::string& GetName() const;

      /**
       * Get the MyOtherClass instance.  Notice the good use of 'const'?
       * @return The instance (could be NULL)
       */
      MyOtherClass* GetObject() const;

   protected:
      ///Protected since the base class is reference counted
      virtual ~MyCoolClass();

   private:
      std::string mName; ///< in-lined member comment
      MyOtherClass* mObject; ///< Some object thing

      /**
       * Could put a long description here.
       */
      bool DoSomeStuff(); ///< Or a short one here
   };
}
#endif // ~MYCOOLCLASS_INCLUDED
```

*** from the mycoolclass.cpp file ***

```
#include <mycoolclass.h>

using namespace dtCore;

////////////////////////////////////////////////////////////////////////
MyCoolClass::MyCoolClass()
: mName("Default")
, mObject(NULL)
{
}

////////////////////////////////////////////////////////////////////////
~MyCoolClass::MyCoolClass()
{
   ///its valid to have a one liner conditional, as long as its surrounded
   ///with braces
   if (mObject != NULL) {mObject == NULL;}
}

////////////////////////////////////////////////////////////////////////
void MyCoolClass::SetName(const std::string& name, bool overwrite)
{
   if (GetName().empty() || overwrite==true)
   {
      mName = name;
   }

   /** I could type up to a maximum of 120 columns of code here if required, but in this
    *  case I don't have to, so I'll try to keep it under 80 columns.
    */
}

////////////////////////////////////////////////////////////////////////
const std::string MyCoolClass::GetName()
{
   return mName;
}

////////////////////////////////////////////////////////////////////////
MyCoolClass* MyCoolClass::GetObject() const
{
   return mObject;
}

///don't forget to leave a newline at the bottom!
```