0 A.D. CODE CONVENTIONS

Dave Loeser Jan Wassenberg

May 7, 2007

Contents

1 Objective

The goal of this document is to provide a standardized coding methodology for the 0 A.D. programming team. With but a few guidelines for layout, commenting and naming conventions, the team should feel as if they are reading their own code when reading someone else's code.

2 Layout

2.1 Formatting

Most editors allow for the conversion of tabs to spaces and most people prefer the use of tabs versus wearing out the spacebar. The size of the tabs is up to each programmer – just ensure that you are using tabs.

Limit the length of a line of code to not more than 80 characters; not everyone has a 1600x1200 display. Functions that have many parameters and extend over 80 characters should be written as:

```
SomeFunction(
    HWND hWnd,
    BITMAP bmDeviceBitmap,
    long lAnimationFrame);
```

instead of:

```
SomeFunction(HWND hWnd,
             BITMAP bmDeviceBitmap,
             long lAnimationFrame);
```

Although the second method is commonly used, it is more difficult to maintain (if the name of the function changed, you would need to re-align the parameters).

2.2 Brackets

Brackets should be aligned, here's an example of good bracket placement:

```
void CGameObject::CleanUp()
{
    if(NULL != m_ThisObject)
    {
        delete m_ThisObject;
    }
}
```

Now we're not out to save vertical lines on the screen; it's about being able to read the code. Therefore, the following style should be avoided:

```
void CGameObject::CleanUp() {
    if(NULL != m_ThisObject) {
        delete m_ThisObject;
    }
}
```

3 Commenting

Commenting is a subject that is sure to cause debate, but minimal comments with maximum expressiveness are preferable. Bad commenting style is shown below:

```
void CGameObject::SetModifiedFlag(bool flag)
{
    m_ModifiedFlag = flag;    // set the modified flag
}
```

The above comment does not tell us anything that we don't already know from reading the code; here's a better approach:

```
void CGameObject::SetModifiedFlag(bool flag)
{
    // This sets the CGameObject's modified
    // flag, which is used to determine
    // if this object needs to be serialized.
    m_ModifiedFlag = flag;
}
```

4 Naming Conventions

4.1 Filenames

Filenames can be freely chosen, but to avoid problems on Unix systems, they should not contain spaces or non-ASCII characters. If the file serves to define one class, e.g. `CEntity`, the file would usually be called `Entity.h`.

4.2 Namespaces

Namespaces are used as a mechanism to express logical grouping.

4.2.1 Global Scope

Symbols belonging to the global namespace should be prefixed with `::`. Example: The Win32 function `::OutputDebugString()` resides in the global namespace and is written with the scope operator preceding the function name.

4.3 Classes

Classes should use concise, descriptive names that easily convey their use.

Classes are named using PascalCase - capitalizing each word within the name visually differentiates them. Example: A class named `CGameObject` is preferred over `gameObject` or `cGameObject`.

4.4 Functions

Functions should use concise, descriptive names that provide innate clues as to the functionality they provide.

Global and member functions should be named using PascalCase. Example: A function named `SetModifiedFlag()` is preferred over `SetFlag()` or `setFlag`.

4.5 Variables

Variable should use concise, descriptive names that provide innate clues as to the data that the variable represents.

Member variables should be prefixed with `m_`, but both `m_camelCase` and `m_PascalCase` may be used according to personal preference (either way, the prefix ensures clarity). Example: `m_GameObject` is more descriptive than `gobj`.

5 Documentation

Each programmer is responsible for properly documenting their code. During code review the code reviewer will ensure that interfaces or APIs are properly documented.

If the comments are formatted in a certain way, they will automatically be extracted and added to the relevant documentation file. It suffices to write them as follows (sample comment for a class):

```
/**
 * An object that represents a civilian entity.
 *
 * (Notes regarding usage and possible problems etc...)
 **/
```

For single-line comments, `///` can be used as well. The comment text is inserted into the documentation, and can additionally be formatted by certain tags (e.g. `@param description` for function parameters). For more details, see the CppDoc documentation.

Each method of a class should be documented as well and here is the suggested method of documenting a member function (continuing with `CExample`):

```
class CExample
{
public:
    CExample();
    ~CExample():
```

(continued)

```
    /**
     * This function does nothing, but is a good example of
     * documenting a member function.
     * @param dummy A dummy parameter.
     **/
    void ShowExample(int dummy);

private:
    intptr_t m_ExampleData;          // Holds the value of this example.
    double m_FairlyLongVariableName; // Shows the lining up of comments
};
```

The ctor and dtor need not be commented — everyone knows what they are and what they do. `ShowExample()`, on the other hand, provides a brief comment as to its purpose. You may also want to provide an example of a method's usage. Member data is commented on the right side and it is generally good (when possible) to line up comments for easier reading.

5.1 Author and Modified By

To promote collective code ownership and encourage making necessary fixes to modules that happen to be written by others, we will avoid explicitly mentioning the author at the top of the file. Such a tag could (subconsciously) be interpreted as "his code only", a condition we want to avoid because "he" might get run over by a bus (thus losing the only source of knowledge and expertise on that piece of code).

On a similar note, we will also avoid modified-by tags. The reasoning is as above, with the additional wrinkle of having to figure out when exactly to consider oneself to have modified the code. Is it after a quick typo fix, adding 2 lines, a function, ...?

Note that the authors of course retain copyright; we can also reconstruct the file history and find out all contributors via SVN revision information. The purpose of this measure is simplicity and improved cooperation and does not deprive anyone of credit.

5.2 Example

Here is a sample header file layout, `Example.h`:

```
/**
 * ========================================================================
 * File        : Example.h
 * Project     : 0 A.D.
 * Description : CExample interface file.
 * ========================================================================
 **/

/*
This interface is difficult to write as it really
pertains to nothing and serves no purpose other than to
suggest a documentation scheme.
*/

#ifndef INCLUDED_EXAMPLE
#define INCLUDED_EXAMPLE

#include "utils.h"

/**
 * CExample
 * This serves no purpose other than to
 * provide an example of documenting a class.
 * Notes regarding usage and possible problems etc...
 **/
class CExample
{
public:
    CExample();
    ~CExample():

    /**
     * This function does nothing, but is a good example of
     * documenting a member function.
     * @param dummy A dummy parameter.
     **/
    void ShowExample(int dummy);
```

```
protected:
    int m_UsefulForDerivedClasses;

private:
    uint8_t m_ExampleData;        // Holds the value of this example.
    int m_RatherLongVariableName; // Shows the lining up of comments
};

#endif    // #ifndef INCLUDED_EXAMPLE
```

From the above we can see that header guards are utilized. Header file comment blocks show filename, project and author; a short overview follows.

The order of declarations ought to be: public followed by protected and finally by private.

6 Standard Template Library

We will make use of the Standard Template Library (STL). Although we may be capable of coding list, maps and queues ourselves and do so more efficiently. Our goal is to create a game not to recreate an existing library.

Having said that, it may make sense to hide some uses of STL objects behind an interface. This can make the code more readable.

7 Singletons

Much debate regarding the use of global variables has been generated over the years so we will not re-enter that discussion. The Singleton design pattern does provide many benefits over that of a pure global variable.

We will make use of the Automatic Singleton Utility as described by Scott Bilas in article 1.3 of the "Game Programming Gems", volume I, "An Automatic Singleton Utility".

8 Strings

A string class has been written, `CStr`, that should be used instead of directly using `std::string` or using C-style strings (i.e. `char*`).