DAZ Script Coding Standards

Below you'll find coding standards that DAZ 3D uses internally. We've provided them here in hopes that you will find them useful in your own endeavors. We find that by using these standards, our code is easier to read and maintain. We hope that they might offer you the same benefit.

Variable Naming

Variable names names should be descriptive, rather than brief. Use the following rules when naming variables:

Notation

Since DAZ Script variables are type-less, use a Hungarian Notation style naming convention to help avoid confusion about what types variables and parameters are expected to be. The following is a list of prefixes for the different basic types:

Prefix	Type
o	Object any variable that is intended to hold an instance of a QObject or DAZ Script custom class besides those listed below.
a	Array type variables
b	Boolean type variables
n	Number type variables
s	String type variables
clr	Color type variables
pal	Palette type variables
pix	Pixmap type variables
date	Date type variables
pnt	Point type variables
rect	Rect type variables
regx	RegExp type variables
size	Size type variables
bytes	ByteArray type variables
font	Font type variables
w	DzWidget type variables
box	DzBox3 type variables
mtx	DzMatrix3 and DzMatrix4 type variables
vec	DzVec3 type variables
quat	DzQuat type variables
rng	DzTimeRange type variables

Variable Name Case

Variable names always start with a lowercase letter, with each subsequent word in the variable name starting with an uppercase letter (no underscores separating words).

Be Descriptive

Always use descriptive variable names. Avoid making variable names from single letter abbreviations that can be difficult to read. One letter variable names should **NOT** be used, except in the following cases:

- i, j, k, n - Should only be used to iterate over a for/while loop where the variables i, j, and k are iterators, and n is the total number of iterations.
- x, y, z, w - Should only be used to represent coordinates in 2D or 3D space.

Global Variables

Global variable names always start with 'g_'. Global variables should be used sparingly, but when they are used they should reside at the top of the file.

Function Naming

Function and method names, like variable names, always begin with a lowercase letter, with an uppercase letter used for every subsequent word in the function name.

Formatting

It is important to format code in a manner that is easily readable.

Indentation

Indentation is a very good visual cue for flow control in a program. Use tabs for indentation (not spaces). The indentation of the code should be increased by one level in each of the following circumstances.

- All code blocks beginning and ending with braces '{ }'.
- All code executed for a conditional statement.
- All code executed in a loop.
- All code inside a try-catch blocks.
- Statements that are continued on the next line.

```
function indentationExample(){
        var i, num = 10;
        var nCount = 20, nRemainder = 0;
```

```
        for( i = 0; i < num; i++ )
                count--;

        if( nCount > 0 ){
                while( nCount > 0 ){
                        nRemainder++;
                        nCount--;
                }
        }

        try {
                while( nRemainder > 0 )
                        print( nRemainder-- );
        }
        catch(...){
                print( "Unhandled exception!" );
        }

        print( "This is a very long string. Okay, it isn\'t that long, "
                "but its long enough that it must be continued on the next
line." );
}
```

Nested Conditional and Loop Statements

In the case of nested conditional and loop statements, braces should always be used to avoid confusion, and avoid logic errors such as the 'dangling else' problem.

Whitespace

A single empty line should be used to visually denote groupings in the code, separating code blocks, algorithms, definition groupings, etc. Multiple consecutive empty lines should not be used.

Commenting

Good commenting habits make the code much easier to maintain. General comments throughout the code make it much easier to debug and extend the code, however, unless written specifically to dissect what is happening in explaining samples, excessive commenting can make the code cluttered and more difficult to follow, and should be avoided - comment briefly, concisely, and whenever necessary - layout the code in blocks that can be commented as a whole, rather than individual lines. Comments that contradict the code are worse than no comments at all. Always make it a priority to keep comments up-to-date when the code changes.

General Comments

General comments should be added to the code wherever appropriate. These may be C style (// Commented) or C++ style (/* Commented */). Brief comments should follow the code on the same line. Comments for blocks of code, or larger comments should precede the code and be at the same indentation level as the code. All algorithms in the code should have a comment briefly describing the function. Larger functions and methods should be blocked out with comments briefly describing the flow of the function.

Readability Comments

Some comments in the code do nothing more than visually separate portions of the code. This can make the code much easier to 'scan' through, and so we have adopted a few comments that serve only as visual cues in the source code.

File Local Sections

These comment blocks are used to separate blocks of file-local global variables, member variables, static variables, etc.

```
/*****************************
   SectionLabel
*****************************/
```

Function Definitions

These comment blocks precede every function definition, and serve to visually separate one function from the next.

```
/*************************************************************************/
```