Virtual Environment Software System (VESS): Coding Standards And Conventions

1-1-2000

Jason Daly
Bryan Kline
Glenn A. Martin

Find similar works at: https://stars.library.ucf.edu/istlibrary

University of Central Florida Libraries http://library.ucf.edu

Recommended Citation

Daly, Jason; Kline, Bryan; and Martin, Glenn A., "Virtual Environment Software System (VESS): Coding Standards And Conventions" (2000). Institute for Simulation and Training. 207.
https://stars.library.ucf.edu/istlibrary/207

This Research Report is brought to you for free and open access by the Digital Collections at STARS. It has been accepted for inclusion in Institute for Simulation and Training by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
Virtual Environment
Software System

Coding Standards and Conventions

Jason Daly, Bryan Kline and Glenn Martin

February 2, 2000
Virtual Environment Software System (VESS)

Coding Standards and Conventions

Jason Daly, Bryan Kline, and Glenn Martin

February 2, 2000

Institute for Simulation and Training
University of Central Florida
3280 Progress Drive
Orlando, FL 32826

IST-TR-00-01
Introduction

The Virtual Environment Software System (VESS) is a suite of libraries developed by the Visual Systems Laboratory at the University of Central Florida's Institute for Simulation and Training (IST). It is based on lessons learned from the Virtual Environment Library (VEL) which was previously developed by IST and used to create the software for various virtual reality applications. The goal of VESS is to provide an application base that is useful and functional using today’s hardware and graphics libraries, extensible to support future hardware and graphics libraries, and easily portable to multiple platforms, graphics systems, and application programming interfaces (API’s).

VESS is designed to simplify and expedite the development of applications where virtual environments are required. It does this by providing a simple interface into the underlying graphics API while integrating support for various input devices, such as joysticks and motion tracking systems, and display devices, such as head-mounted displays and shutter glasses. Additionally, VESS provides behaviors and motion models to allow the user to manipulate his or her viewpoint as well as control and interact with objects in the virtual environment. The user’s viewpoint can be independent or attached to any transformable object in the scene. Other useful routines such as collision detection and terrain following are also provided.

VESS is designed for easy portability. Its multi-layered architecture allows the developer to focus on the details of the application, without worrying about the specifics of the graphics API or hardware interfaces. Thus, applications built using the VESS libraries will be easily portable to any other supported platform. Currently, VESS runs on IRIX and Linux platforms using the SGI Performer API. Other platforms and APIs will be supported in the future.

Purpose of this document

This document details the conventions and practices used in the VESS source code. This coding standard was adopted to ensure consistency between the various VESS modules, and to maximize the readability and modifiability of the libraries. This document is intended primarily for IST employees who need to make changes and/or additions to VESS. Developers may also find this information useful when perusing the VESS source code and designing applications.
Naming conventions

VESS identifiers, symbolic constants, macros, class names, and function names will follow these guidelines for their names.

Variable Names

Variable names will be lowercase with the first letter of each word after the first word capitalized. Variable names, especially function parameters, should be descriptive and of sufficient length to foster readability. Loop control variables need not be lengthy or descriptive provided the variable is used only to index arrays. Global variables should be kept to a minimum, but where they are used, the name will also be lowercase with the first letter of each word (including the first word) capitalized. All global variables will begin with a “vs” prefix.

Example variable names:

```c
int anExample;
char example;
double yetAnotherExample;
long vsGlobalVariableExample;
```

Acceptable loop:

```c
int i;
for (i = 0; i < maxCount; i++)
{
    // It is immediately clear that i is simply indexing an array
    anArray[i] = someValue;
}
```

Unacceptable loop:

```c
int i;
i = 10;
while (i > 0)
{
    // The use of i in this loop is not obvious, so i should be renamed
    i -= 3;
    i = checkValue(i);
}
```

Class Names

Class names will be lowercase with the first letter of each word capitalized. The prefix “vs” will precede all class names. Class names should be sufficiently descriptive of their purpose and function, but not overly verbose.
Examples:

class vsInputDevice

class vsMatrix

class vsWindow

**Function names**

Class member function names will be lowercase with the first letter of each word after the first word capitalized. As with global variables, non-member functions should be kept to a minimum. Their names will also be lowercase with the first letter of each word (including the first word) capitalized, and their names must begin with the "vs" prefix.

Functions should be formed of a verb-noun pair describing their function whenever possible. For member functions, a single verb can be used in place of a verb-noun pair if the operation being performed deals primarily with the object itself and adding the noun would be superfluous. Functions returning boolean values should use a form of "be" as their verb (e.g. "is" or "was").

Examples:

```cpp
vsDoSomething();
theWindow->addPane(aPane);
if (button->isPressed())
  doSomething();
geraphics->update();
vector->add(anotherVector);
someList->addElement(anElement);
```

**Symbolic constants, enums, and macros**

All constants and macros will be uppercase with underscores separating words. A "VS_" prefix must precede the constant/macro name.

Examples:

```cpp
#define VS_MAX_PIPES 3
#define VS_SQR(x) ((x) * (x))
enum
{
  VS_X = 0,
  VS_Y = 1,
  VS_Z = 2,
  VS_W = 3
}
**Files**

VESS source and header file names will match the class they define or implement. There will only be one class defined or implemented in each file (supporting struct’s and other items relating to the class are permitted). On UNIX platforms, files will end with “.c++” for source files and “.h++” for header files. On Microsoft Windows platforms, files will end with “.cpp” for source file and “.hpp” for header files. Files that don’t define or implement classes must be justified and should be kept to a minimum. These files will follow the naming convention for classes and the preceding rule for extensions.

Configuration file names should be descriptive of the kind of configuration they are handling (alpha.cfg is rather cryptic, for example). Configuration files used exclusively by VESS will begin with a “vs” prefix. All configuration files will end in a “.cfg” extension.

**Examples:**

vsInputDevice.h++
vsWindow.c++
vsGlobals.h++
vsInput.cfg
Formatting Conventions

VESS source and header files will follow the following formatting guidelines.

Header file organization

VESS header files will begin with a standard header including copyright notice, license information, contact information, and a description block. The details of this header appear in a later section.

Guard bars will appear following the header setting of the remainder of the header file. Specifically,

```c
#ifndef VS_HEADER_FILE_HPP
#define VS_HEADER_FILE_HPP
```

will appear at the beginning of the file after the standard header, and

```c
#endif
```

will appear at the end of the file. The symbol used in the guard bars will match the name of the header file with all uppercase letters and underscores between each word, prefix, and extension.

Within each header file, all include directives, symbolic constants, enum blocks, macros, typedefs, global variables, and supporting structures will be arranged into groups and not interleaved with each other. All of these will appear before the class definition.

The class definition will be the last construct in the header file (except for the bottom guard bar).
Class definitions

The first line of a class definition will contain the "class" keyword, the class name and the inheritance instructions. Components will be separated by a single space. For example:

class vsExampleClass : public vsParentClass

The class definition's opening brace appears in the first column on the line immediately following the first line of the definition. The closing brace appears on the line immediately following the last element in the definition, followed by the semicolon.

Access descriptors will begin in the first column. Besides the standard private, protected, and public access descriptors, a descriptor named VS_INTERNAL will be used. VS_INTERNAL denotes functions that are publicly accessible, but intended for use by other VESS classes only (not by applications). The access descriptors will appear in this order: private, protected, VS_INTERNAL, and public. There will be a blank line immediately after each access descriptor. There will also be a blank line before each access descriptor except the first one.

Function and variable names should appear on the same line as their corresponding type unless placing the type and function name on the same line would exceed column 80. Types will always be indented four spaces. The further indentation of variable and function names is determined by the following guidelines:

- Variable and function names will be left-aligned with each other within each access descriptor block.
- Variable and function names will be indented a minimum of 16 spaces.
- Variable and function names will appear a minimum of four spaces after the type.

If a function's parameter list is too long to fit on a line, the remainder of the list will appear left-aligned with the first parameter in the list. If this indentation does not allow sufficient space for one or more of the parameters, the longest parameter in the list will be right-aligned with the first line of the parameter list, and the remaining parameters will then be left-aligned with the longest parameter. Always divide the parameter list after a comma.

Keywords such as "virtual" and "static" are considered part of the type, and the pointer "*" character is considered part of the variable or function for formatting purposes.

Within each access descriptor block, variables will appear first followed by functions. No variables should appear in the public or VS_INTERNAL blocks (i.e. no public variables are allowed). Related variables and related functions should be grouped with groups separated by a single blank line.

No source code will appear in the header file. This means no functions will be defined inline, regardless of how simple. Every function that is declared in a header file should be defined in the corresponding source file (except for pure virtual functions).

Exceptions can be made to the preceding rules for exceedingly long type names, function/variable names, or parameter lists if doing so significantly improves readability. Each declaration not following the preceding rules will appear in its own group.

If any declaration is to be commented, the comment will appear on the line(s) immediately before the declaration, left-aligned with the type (i.e., indented four spaces). Line comments, not block comments, will be used.

Forward declarations should be avoided whenever possible.
Friend class declarations and friend methods will be avoided unless absolutely necessary. The design and implementation of the related code will be rethought before friend classes and/or methods will be permitted.

Example class definition:

class vsExampleClass : public vsParentClass
{
    protected:

        // Comment goes here
        int example1;
        int example2;
        double example3;

        // This declaration breaks the rules to avoid making the rest
        // of the declarations indent exceedingly far
        AReallyReallyReallyReallyLongTypeName *example4;

    public:

        vsExampleClass(int param1, int param2);
        vsExampleClass(int param1, int param2, int param3,
            double reallyReallyReallyReallyLongParam,
            char param5);
        virtual ~vsExampleClass(void);

        virtual int getNumThingys(void);
        virtual void setNumThingys(int newNumThingys);

        // This declaration breaks the rules like the variable declaration
        // above. It also breaks the four-space rule to allow this
        // declaration to fit on one line (only a three-space indentation
        // after the type name)
        virtual AReallyReallyReallyReallyLongTypeName *getThingy(int id);

        // The parameter list is too long for the line
        virtual void setThingy(
            AReallyReallyReallyReallyLongTypeName thingy);
};
Struct and enum blocks

struct{} and enum{} blocks begin with "struct" or "enum" followed by the name (if the block is named) on the first line. The opening brace appears on the next line, and the closing brace appears on the line immediately after the last element of the block, followed by the semicolon.

Within structs, each type will be indented four spaces. Each variable name should be indented a minimum of four spaces, and all variable names should be left-aligned within the struct block.

Within enums, the symbols should be indented four spaces. Values can be explicitly defined if desired. If so, the equals sign will appear a minimum of one space after the symbol name, and all equals signs will be column-aligned. The values will appear one space after the equals sign.

Related elements of a struct should be grouped with a single blank line separating groups. Enums should not include more than one set of related elements, and thus, would not have more than one group.

Symbolic constants and macros

Related symbolic constants will be grouped together with a single blank line separating groups. The constant’s name will appear one space after the #define directive. The constant’s value will appear at least one space after the constant’s name. If the constant is part of a group, the values in that group will be left-aligned with each other.

The use of macros should be kept to a minimum. The name of a macro will appear one space after the #define directive. The macro definition will appear four spaces after the macro name.

Examples:

```
#define VS_TRUE 1
#define VS_FALSE 0

#define VS_SQR(x) ((x) * (x))
```

Source file organization

Source files will follow a similar organization to header files. All include directives, symbolic constants, macros, typedefs, global variables, and supporting structures will be grouped separately and not interleaved with each other. All of these constructs will appear before any method definitions.
**Source code formatting**

Function definitions will appear in the same order declared in the header file. They will begin with a standard comment briefly stating the function's purpose and operation. Exceedingly long comments should be avoided. The sequence of operations can be mentioned, but must not be overly verbose. Save the details for the documentation.

The comment will use C++ style line comments and not block comments. The comment will begin and end with a line containing the comment operator followed by a single space and then 72 hyphens. The lines of text between these header/footer lines will not exceed the header/footer lines' length. See the example below.

After the header comment, the first line will contain the function return type, class name, function name, and parameter list. If the entire parameter list will not fit on the first line, use the format specified in the class definition section to break up the parameter list.

Local variables should all be defined at the top of the function. In general, variables defined in the middle of code should be avoided. If the list of local variables is extensive (e.g.: ten or more), related variables should be grouped with a single blank line separating groups.

The highest level of braces in each function definition will appear in column 1. The opening brace will appear immediately after the last line of the parameter list. The closing brace will appear on the line immediately following the last line of the function definition's source code (no blank lines in between). No semicolon will appear after the final brace of the function definition.

The standard indentation for source code is four spaces. Each nested block of code will be indented four spaces further from the indentation of its parent block. Braces for each nested block will appear in line with the parent block's indentation (flush left).

The parameter lists in function definitions and function calls will include a single space after each comma and no spacing after the opening parenthesis or before the closing parenthesis. In function definitions, there will only be a single space between the type name and the parameter name.

Binary operations will be formatted with a single space between the operator and operands. Unary operators will not have this space. See the first line of the for loop below for examples of this rule.

Complex parenthetical statements (statements with multiple levels of parentheses) should be spaced and formatted to maximize readability. Use judgment on a case-by-case basis to determine the spacing for these statements.

If a line of code is too long to fit within 80 columns, the line must be divided and the remainder indented four spaces from the beginning of the line above. For string constants exceeding this limit, divide, the constant with closing quotation marks at the end of the line, and an additional set of opening quotation marks indented four spaces from the beginning of the first line.

When using the printf statement to print to standard out, use one printf statement per line of output. When skipping lines, a single printf statement can be used to skip multiple lines, as long as no other output appears within that statement. No printf statement should exceed 80 columns of output. Error messages should be preceded by a prefix indicating the class and function name in which the error occurred. See below for an example of this prefix.
Any commenting of functions beyond the standard header comment will also use C++-style line comments and not block comments. Comments should be inserted where the procedural or logical flow of the code is not clear (as in multiple nested loops and complex conditional structures), or when complex operations are carried out (as in intensive mathematical operations). Comments should be inserted before the operation(s) to which they refer, and should have the same indentation as the rest of the code in that block.

Example:

```
// A function to take arbitrary parameters of various types and perform
// a cryptic set of operations on them, returning a useless integer value

int vsExampleClass::getUselessInformation(char *stringInfo,
                                          int *intInfo, double *doubleInfo)
{
    int aVariable;
    int i;
    aVariable = 0;
    for (i = 0; i < *intInfo; i++)
    {
        // Comments should appear like this
        doubleInfo[i] = ((aVariable + 0.0543) * ((i + 3.434) * 0.3));
        aVariable *= (int)doubleInfo[i];
    }
    if (aVariable > VS_MAX_VALUE)
    {
        printf("vsExampleClass::getUselessInformation: Value too "
               "large. Something bad might happen.\n");
        printf("\n\n");
    }

    return aVariable;
}
```
Other code considerations

Unconditional branches (goto and setjmp/longjmp) will be avoided at all costs. If an error trapping or other contingency mechanism is required, it must be implemented with C++ exception handling routines (throw, try, and catch). If this is not possible, the implementation must be rethought before goto or setjmp/longjmp will be permitted.

The statement

\[ a += b; \]

is preferred over

\[ a = a + b; \]

The question/colon operator is to be avoided. Use if/then/else instead.

Pointer variables, not variable references will be used to pass parameters by reference. This is to keep the method of parameter passing consistent, thus maximizing usability.