Coding Standards
Hive Assault Project

Introduction

Why Standardize?
Trying to read code that someone else has written is a difficult task as anyone who has done it can probably attest to. The style that code is written in can make a big difference as to how readable the code is to everyone else. A uniform coding style makes it much easier for new coders to jump into the code and get up to speed quickly.

Is there anything bad about it?
Yep. Usually, people are quite religious when it comes to coding style. If it’s not your style, it’s usually considered to be stupid. It’s like driving; the people who are going faster than you are jerks, and the people going slower are idiots. Regardless, people will have to overcome their personal preferences for the good of the project. Plus, there’s the common enemy who came up with these standards. Let’s call him… oh…. Bob.

Coding Style
We’ll try to adhere to the guidelines set by Sun for Java since most people are familiar with them.

Names

Functions should always start with a lower case letter with subsequent words separated by capital letters. For example,

/**
 * A function that does something.
 * @param a doesn’t matter
 * @param b doesn’t matter
 */
void doSomethingNow ( int a, int b );

Variables should always start with a lowercase. Ex.

/// the grid for the fluid simulation
int** grid;

Class names should always follow the rules for functions except they should start with a capital letter. Ex.

class FooBar { … };
Constants

There should be no “magic numbers” in the code. If you use a constant in your code, take it out and make it a variable. Programmers tend to reuse constants a lot in their code and it makes it difficult to rewrite all of the constants when they change. Ex.

```c
void doSomething ( void )
{
    int i;

    for ( i = 0; i < 30; i++ )
    {
        x = width / 30;
        y = height / 30;
        ...
    }
}
```

This can be rewritten as

```c
void doSomething ( void )
{
    int i;
    const int GRID_WIDTH = 30;

    for ( i = 0; i < GRID_WIDTH; i++ )
    {
        x = width / GRID_WIDTH;
        y = height / GRID_WIDTH;
        ...
    }
}
```

Constants should always be written using capital letters with underscores for separation between words. They should either be compiler defines ( #define GRID_WIDTH 30 ) or using the const directive as above. This is alleviates problems when two “magic numbers” happen to be the same, but only one needs to be changed in the code. That’s a nightmare.
**Classes**

The different parts of a class should follow a specific order within each type of access (public, private, etc…). This makes it easier to find things inside of a class quickly. The order should be as follows:

```
class FooBar
{
    // constants

    // fields (non-constant data members)

    // constructors/destructors

    // methods
}
```

**Brackets**

Several things need to be said about brackets. First, always use brackets! Don’t be lazy. For example,

```
if ( a == 5 )
    b = 12;
    c = 13;
```

This code is confusing. It should be rewritten as

```
if ( a == 5 )
{
    b = 12;
}
    c = 13;
```

Ah, much better. Don’t get lazy with brackets. It makes it much easier to go back and add code into the consequence portion of the if-statement later if there are brackets already in place. This isn’t just for if-statements but for all statements that can omit brackets.

The second thing about brackets is where they should be placed. Bob thinks that brackets should be placed on a new line. This is so that code within each scope is easily identified. It also makes it quite easy to balance brackets when coding. For example,
// brackets always start on a new line
void doSomething ( void )
{ // <- notice bracket here
  if ( a == 5 )
  { // <- notice bracket here, not on the line above
    b = 12;
  }
  else
  { // <- here’s another one
    b = 13;
  }
}

as opposed to

void doSomething ( void ) {
  if ( a == 5 ) {
    b = 12;
  } else {
    b = 13;
  }
}

The second approach seems more confusing with respect to scope.

If Statements

Only one thing really needs to be said about if-statements: DON’T USE THE SHORTCUT VERSION! It’s ugly and unreadable. Ex.

a == 5 ? b = 12; : b = 13;

What does this do? I’ve seen three or four of these things nested into one statement before. I still don’t know what it did. Don’t be lazy. Write it so that it’s readable. Short and compact code is not always readable code.

if ( a == 5 )
{ 
  b = 12;
}
else
{ 
  b = 13;
}
**Spacing/Tabs**

Lines should not exceed 80 characters. Otherwise, it makes the code impossible to print and difficult to read if the programmer must scroll horizontally back and forth. Try to break up long lines on natural boundaries. Ex.

```c
// imagine that this is a long line
if ( a == 5 &&
    b == 12 &&
    c != 3 )
{
...
}
```

Code should also be indented with tabs. This has both good and bad consequences. The code can be indented to a user’s preference by changing the tab spacing. Suggested tab spacing is 3 or 4 units. However, the code’s appearance is not independent of the viewer as it would be if spaces were used. In the end, tabs are chosen due to Visual Studio’s preference towards using tabs and our suggestion of using Visual Studio as the IDE of choice for this project.

Spaces should be used to separate tokens for clarity. It makes the code much more readable than cramming everything together. Spacing always makes the code more readable so don’t be afraid to hit the space bar when coding. For example,

```c
a=((b+3)*53+4);
```

rewrite as

```c
a = ( ( b + 3 ) * 53 + 4 );
```

**Documentation**

“If your program isn’t worth documenting, it probably isn’t worth running.” (Nagler 1995)

Most everyone on this project is familiar with the JavaDoc style of documenting code. Unfortunately, we’re not using Java; we’re using C/C++. After surveying the available documentation tools available for C/C++, Bob decided on a program called Doc++. It actually does documentation for Java, C, and C++. The documentation style is modified for C/C++ a bit, but it is quite similar to JavaDoc’s style and, most likely, you’ll never see the differences.

The headers of the files must always be documented. Here’s an example:
#ifndef MY_H_FILE
#define MY_H_FILE

#include <math.h>

/**
 * This is an example of how to document a file. This class
 * does nothing.
 * 
 * @author Bob Smith
 * @version 1.0
 */

class MyClass {
    public:
        /// this data member is an x coordinate
        float x;
        /// this data member is a y coordinate
        float y;

        /**
         * This function adds the given values to x,y
         * 
         * @param dx the value to add to the x-coordinate
         * @param dy the value to add to the y-coordinate
         * 
         * @return 1 if the operation succeeds or 0 otherwise
         */
        int addStuff ( float dx, float dy );
};

#endif

In JavaDoc, the /** starts a JavaDoc style comment. Doc++ retains this ability, but it can also start one line comments with a triple / (///) as can be seen above for the member variable declarations. Further information on Doc++ can be found at http://www.zib.de/Visual/software/doc++/index.html
Not only does the header file need to be documented, but the rest of the code needs documentation as well. Don’t just describe the function in your documentation, but why you do certain things. Explain the algorithm that you’re coding. Try to make the code so that someone else could understand it easily. If you leave out certain features or want to code something differently but don’t have time, document it. “// TODO” is great.

```c
void myFunction ( void )
{
    int a = 5;

    b = a * 4;
    // TODO: finish this function, the Simpsons came on so I stopped
}
```