Crown Code Conventions (rev 1.2)

1. File names
   1.1 File suffixes
   1.2 Common file names
   1.3 Text file type

2. File organization
   2.1 Source files
   2.1.1 Beginning comments
   2.1.2 #pragma once directive
   2.1.3 Include statements
   2.1.4 Class declarations

3. Indentation
   3.1 Line width
   3.2 Wrapping lines

4. Comments
   4.1 Implementation comment format
   4.1.1 Block comments
   4.1.2 Single-line comments
   4.1.3 Trailing comments
   4.1.4 End-line comments
   4.2 Documentation comments

5. Declarations
   5.1 Number per line
   5.2 Initialization
   5.3 Placement
   5.4 Class declarations

6. Statements
   6.1 Simple statements
   6.2 Compound statements
   6.3 return Statements
   6.4 if, if-else, if else-if else Statements
   6.5 for Statements
   6.6 while Statements
   6.7 do-while Statements
   6.8 switch Statements
   6.9 try-catch Statements

7. White space
   7.1 Blank lines
   7.2 Blank spaces

8. Naming conventions

9. Programming practices
   9.1 Providing access to instance and class variables

1 File Names

This section lists commonly used file suffixes and names.

1.1 File Suffixes

Crown uses the following file suffixes:

File type	File suffix
Source files	.cpp
Header files	.h

1.2 Common File Names

Frequently used file names include:

Makefile	The preferred name for makefiles.
README	The preferred name for the file that summarizes the contents of a particular directory.
TODO	The preferred name for the file that summarizes tasks to be accomplished.

1.3 Text File Type

Format for text files is the Unix one (i.e. lines end only with a line-feed character). Adjust your favourite editor in order to save in that format.

2 File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

2.1 Source Files

Each Crown source file contains a single class. **The file name must exactly match the name of the class it contains** except rare cases where it is not possible due to pratical reasons. (Conflicting file names, compiler troubles etc.)

Crown source files have the following ordering:

- Beginning comments
- #ifndef - #define - #endif statements (only for header files)
- Include statements
- Namespace statements
- Forward declarations
- Class declarations
- Blank line

2.1.1 Beginning Comments

TODO

2.1.2 #pragma once directive

In order to avoid any linking-related error, every header file must exlude itself from multiple inclusions through the #pragma once directive. It is non-standard but widely supported.and offers several advantages over #include guards.

```
#pragma once
...
```

2.1.3 Include Statements

The first non-comment line of most Crown source files is a `#include` statement. Absolute include paths come before relative ones. After that, other `#include` statements can follow in alphabetic order. At the end of #include list comes `namespace` statements and then forward declarations in alphabetic order. For example:

```
#include <GL/glew.h>
#include “Config.h”
#include “Vec2.h”
#include “Vec3.h”

namespace Crown
{

class Mat3;
class Mat4;
class Quat;
```

```
…

} // namespace Crown
```

2.1.4 Class Declarations

The following table describes the parts of a class declaration, in the order that they should appear.

Part of Class Declaration	**Notes**
Class documentation comment (`/**...*/`)	
`class` statement	
Class implementation comment (`/*...*/`), if necessary	This comment should contain any class-wide information that wasn't appropriate for the class documentation comment.

Utils typedefs	Often when using templated classes such as List<Type> or Dictionary<Type>, a list of private typedefs prevents typing injuries and simplifies code reading/maintenance. Use whenever template classes come in play.
	Example:
	`typedef List<Entity*> EntityList;`
	`typedef Dictionary<Foo, Bar> FooBarDict;`

Constructors	
Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.
Static methods	
Instance variables	First `public`, then `protected`, and then `private`.
Static instance variables	
Friend classes	

3 Indentation

Four spaces should be used as the unit of indentation. Use only tabs for indentation.

3.1 Line Length

Avoid lines longer than 130 characters.

3.2 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line using tabs.

Here are some examples of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
      longExpression4, longExpression5);

var = someMethod1(longExpression1,
            someMethod2(longExpression2,
                  longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 – longName5)
                  + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                              - longName5) + 4 * longname6; // AVOID
```

Here is an example of method indentation:

```
    // CONVENTIONAL INDENTATION
    SomeMethod(int anArg, Object anotherArg, String yetAnotherArg,
               Object andStillAnother)
    {
        ...
    }
```

4 Comments

Crown source files can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /*...*/, and //. Documentation comments are those found in Doxygen documentation. :D

Implementation comments are meant for commenting out code or for comments about the particular implementation. Documentation comments are meant to describe the specification of the code, from an implementation-free perspective. to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.
Comments should never include special characters such as form-feed and backspace.

4.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

4.1.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

4.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 4.1.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Crown:

```
if (condition)
{
      /* Handle the condition. */
      …
}
```

4.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in Crown's code:

```
if (a == 2)
{
      return TRUE;            /* special case */
}
else
{
      return IsPrime(a);      /* works only for odd a */
}
```

4.1.4 End-Of-Line Comments

The `//` comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1)
{
      // Do a double-flip.
      ...
}
else
{
      return false;     // Explain why here.
}
//if (bar > 1)
//{
//    // Do a triple-flip.
//    ...
//}
//else
//{
//    return false;
//}
```

4.2 Documentation Comments

TODO

5 Declarations

5.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level;  // indentation level
int size;   // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int         level;           // indentation level
int         size;            // size of table
Object      currentEntry;    // currently selected table entry
```

5.2 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

5.3 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void myMethod()
{
      int int1 = 0;          // beginning of method block

      if (condition)
      {
            int int2 = 0;    // beginning of "if" block
            …
      }
}
```

The one exception to the rule is indexes of `for` loops, which in C++ can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
…

myMethod()
{
      if (condition)
      {
            int count = 0;   // AVOID!
            …
      }
…
}
```

5.4 Class Declarations

When coding Crown classes, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the very next line after the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample : public Object
{

public:
```

```
        Sample(int i, int j)
        {
              ivar1 = i;
              ivar2 = j;
        }

        int emptyMethod() {}

        …

    private:

        int ivar1;
        int ivar2;
    }
```

- Methods are separated by a blank line

6 Statements

6.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++;           // Correct
argc--;           // Correct
argv++; argc--;   // AVOID!
```

6.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

6.3 return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return MyDisk.GetSize();

return (size ? size : defaultSize);
```

6.4 if, if-else, if else-if else Statements

The `if-else` class of statements should have the following form:

```
// if
if (condition)
{
      statements;
}

// if-else
if (condition)
{
      statements;
}
else
{
      statements;
}

// if-else-if
if (condition)
{
      statements;
}
else if (condition)
{
      statements;
}
else
{
      statements;
}
```

Note: `if` statements always use braces {}. Avoid the following error-prone form:

```
if (condition)          //AVOID! THIS OMITS THE BRACES {}!
     statement;
```

6.5 for Statements

A `for` statement should have the following form:

```
for (initialization; condition; update)
{
      statements;
}
```

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update) ;
```

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

6.6 while Statements

A `while` statement should have the following form:

```
while (condition)
{
      statements;
}
```

An empty `while` statement should have the following form:

```
while (condition) ;
```

6.7 do-while Statements

A `do-while` statement should have the following form:

```
do
{
      statements;
}
while (condition);
```

6.8 swith Statements

A `switch` statement should have the following form:

```
switch (condition)
{
      case ABC:

           statements;
           /* falls through */

      case DEF:

            statements;
            break;

      case XYZ:

            statements;
            break;

      default:

            statements;
            break;
}
```

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

6.9 try-catch Statements

A `try-catch` statement should have the following format:

```
try
{
      statements;
}
catch (ExceptionClass e)
{
      statements;
}
```

7 White Space

7.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block (see section 4.1.1) or single-line (see section 4.1.2) comment
- Between logical sections inside a method to improve readability
- At the end of every source file

7.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

  ```
  while (true)
  {
        ...
  }
  ```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except `.` should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

  ```
  a += c + d;
  a = (a + b) / (c * d);

  while (d++ = s++)
  {
  ```

```
        n++;
}

PrintSize("size is " + foo + "\n");
```

- The expressions in a `for` statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space. Examples:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

8 Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	**Rules of Naming**	**Examples**
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	`class Device;` `class RenderWindow;`

Methods	Methods should be verbs, in mixed case with the first letter uppercase, with the first letter of each internal word capitalized.
	Every method not intended to end-user usage (i.e. that only engine classes should use internally) must start with underscore _.
	Every read-only method (i.e. every method that will not change the value of its class members) must be flagged as const.
		`Run();`
		`GetMatrix();`
		`_EndUsersShouldNotUseMe();`
		`int ImReadOnly() const;`

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _.
	Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`,
		`int i;`
		`char c;`
		`float myWidth;`
		`class Camera`
		`{`
		`public:`
		`…`
		`private:`

	and `n` for integers; `c`, `d`, and `e` for characters.
	All variable names inside a class must start with 'm' character.
		`    float mAspect;`
		`    float mFOV;`
		`    …`
		`};`

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	`const int MIN_WIDTH = 4;` `static const int MAX_WIDTH;`

Enums	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
	Because of Enums are collections of constants, their elements' names follow Constants' rules except the fact they must start with a word made up of all capitalized letter in Enum's name.
		`enum PixelFormat`
		`{`
		`    PF_RGB8 = 0,`
		`    PF_RGBA8,`
		`    …`
		`};`

9 Programming practices

9.1 Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class, then it's appropriate to make the class's instance variables public.

9.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
ClassMethod();                //OK
AClass::ClassMethod();        //OK
anObject.ClassMethod();       //AVOID!
```

9.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

9.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) // AVOID!
{
      …
}
```

should be written as

```
if ((c++ = d++) != 0)
{
      …
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r; // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

9.5 Miscellaneous Practices

9.5.1 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)         // AVOID!
if ((a == b) && (c == d))     // RIGHT
```

9.5.2 Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression)
{
      return true;
}
```

```
else
{
	return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition)
{
	return x;
}

return y;
```

should be written as

```
return (condition ? x : y);
```

9.5.3 Expressions before `?' in the Conditional Operator

If an expression containing a binary operator appears before the `?` in the ternary `?:` operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x;
```

9.5.4 Special Comments

Use `XXX` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken. Use TODO to flag something that is partially or not already ready to use.

10. Code Examples

10.1 Crown Source File Example

```
#pragma once

#include "EventDispatcher.h"
#include "Renderer.h"
#include "RenderWindow.h"
#include "List.h"

namespace Crown
{

class SomeClass : public Foo
```

```
{

	typedef List<Renderer*> RendererList;

public:

	//! Constructor
	SomeClass();

	//! Destructor
	~SomeClass();

	//! Returns whether is visible
	void IsVisible() const;

	//! Sets whether is visible
	void SetVisible();

	//! Returns value of blah
	float GetBlah() const;

	//! Sets value of blah
	void SetBlah();

	//! System method
	void _EndUsersShouldNotUseMe();

	//! Returns something
	int ImReadOnly() const;

	//! Returns the static integer's value
	static int* GetStaticInteger();

private:

	bool mVisible;
	float mBlah;

	static int mStaticInteger;

	friend class Bar;
};

} // namespace Crown
```

11 List Of Changes

Changes since 1.0:

Changed curly bracket style.

Changes since 1.1:

Use #pragma once instead of #include guards.