The Game Architecture 2018 Coding Standard

This document defines the coding standard for the Game Architecture course. Following this coding standard is required to receive full credit on homework assignments.

The purpose of this, and more generally, any coding standard is two-fold:

1. Discourage bad practice and encourage good practice. This directly results in more robust and efficient code.
2. Enforce consistency across a codebase. While the look and feel specified in a coding standard is often subjective, consistency aids in readability and maintainability; two critical components of a successful production codebase.

Good Practice

- Unless otherwise indicated, feel free to use features of C++11.
- Treat compiler warnings as errors. Code should be modified to avoid warnings.
- When passing data of a type with non-trivial constructors or size greater than 8 bytes to a function, use a pointer or reference to avoid expensive copy costs.

```
void foo_function(ga_big_type data) {} // BAD

void foo_function(const ga_big_type* data) {} // Good
```

- Mark pointers as const when read-only. Mark methods as const when read-only.
- Mark methods as override when overriding a virtual in a base class.
- Avoid unnecessary heap allocation. Hoist memory allocations outside loops if possible.
- Avoid runtime type information.

```
auto derived = dynamic_cast<ga_derived*>(base); // BAD
```

- Avoid exceptions.

```
throw ga_failure(); // BAD
```

- Declare variables in the tightest possible scope.

```
 // BAD

    ga_foo* foo;
    if (...)
    {
          foo = get_foo();
          foo->do_something();
    }
// Good
if (...)
{
    ga_foo* foo = get_foo();
    foo->do_something();
}
```

- Use constants or enums instead of magic numbers.

```
        float area = 2.0f * 3.141592f * radius * radius; // BAD

        float area = 2.0f * k_pi * radius * radius; // Good
```

- Use type “int” for integers by default. Use other integers types only when it adds meaning or a different range of values is important.

- Use 32-bit floating point type “float” for floating point values unless the range or precision of 64-bit doubles is required.

```
        double area = 2.0 * k_pi * radius * radius; // BAD

        float area = 2.0f * k_pi * radius * radius; // Good
```

- Write comments that explain why code is the way it is. Use correct grammar.

```
        // BAD:

        // increment i
        i++;
```

- Add class header comment for all public classes.

```
        /*

        ** Entity object.
        ** A bucket of components. No classes should derive.
        ** All functionality should be in components.
        ** @see ga_component
```

```
*/
class ga_entity final {};
```

- Minimize the number of #includes. Try to avoid including system headers inside other headers by forward declaring types.
- Do not check-in large blocks of commented out code, just delete them.
- Try to limit line length to about 80 characters.
- Refactor or break functions to avoid indenting more than 4 levels.
- Clean up resources allocated by a class in its destructor.

Consistency

- When in doubt, write code in the prevailing style of the module you are editing.
- Use hard tabs, not spaces.
- When naming identifiers, use snake case.

```
bool ImportantFlag; // BAD

bool important_flag; // Good
```

- When naming identifiers intended to be part of public API, prefix with “ga_”.

```
struct some_data; // BAD

struct ga_some_data; // Good
```

- When naming constants and enums, prefix with “k_”.

```
const int meaning_of_life = 42; // BAD

const int k_meaning_of_life = 42; // Good
```

- When naming member variables of a class or struct, prefix with underscore.

```
class ga_foo

{

	int bar; // BAD

	int _bar; // Good
};
```

- Braces appear alone or with a semicolon, on new lines.

```
// BAD

if (something()) {
}

// Good
if (something())
{
}
```

- Always use braces for statements following conditionals and loops. Add a space between control statement and the parentheses that follow it.

```
// BAD
if(is_active()) do_it();
// Good
if (is_active())
{
      do_it();

}
```

- Add spaces around operators.

```
float area=2.0f*k_pi*radius*radius; // BAD

float area = 2.0f * k_pi * radius * radius; // Good
```

- Add line breaks after operators, not before.

```
// BAD

if (test1()
      && test2()
      && test3()
      && test4()
      && test5())

{
}
```

```
// Good
if (test1() &&
      test2() &&
      test3() &&
      test4() &&
      test5())
{
}
```

- Avoid more than 3 consecutive blank lines.