NTNU | Department of Computer Science

PROG2900 - Bachelor Thesis

Code Convention Document

Lokførerskolen Sim v2

Authors:
Thomas Arinesalingam
John Ole Bjerke
Endre Heksum
Henrik Markengbakken Karlsen

February, 2022

Contents

1 Introduction

With the goal of producing a highly understandable and intuitive code base, we declare this document to define the standard coding practices used in this project. Consistent naming conventions, documentation standards and layout formatting are essential for ensuring and maintaining professionalism with emphasis on readability. The standards defined in this document are inspired by the established standard documentation for Unreal Engine, as written by Epic Games[1].

2 Naming Conventions

2.1 Classes

All classes should follow the PascalCase naming convension where each word in the variable name should start with a capital letter. Example:

```
class PlayerAbilities {

}
```

The only exeptions to this is the special Unreal Engine Class inheritance naming listed below:

- Classes which inherits from UObject are prefixed by `U`.
- Classes which inherits from AActor are prefixed by `A`.
- Classes which inherits from SWidget are prefixed by `S`.
- Classes that are abstract interfaces are prefixed by `I`.

2.2 Methods

The naming of methods should be used to describe the effect of the method. Or describe the return value if the method has no effect. Methods should follow the PascalCase naming convension.

One exeption is for functions that returns boolean variables. these methods should always ask a true/false question such as:

```
HasScored();
ShouldEndGame();
```

[1] https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/

2.3 Variables

Variable names should follow the camelCase naming convension, meaning the first word of the variable name should be all lower case letters, all following words in the name should start with a capital letter. The words used should be nouns and descibe the variable. This practise supports the variable names:

```
// Good naming
int    scoredGoals = 3;
string playerName = "Maradonna";
```

but not the variable names:

```
// Bad naming
int    scored_goals = 3;
string PlayerName = "Maradonna";
```

The letters in constant names should be all upper case:

```
const int MAXGOALS = 32;
```

3 Documentation

3.1 Commenting

- Comments should be written in U.S. English.
- All code should be self-documenting. If code is considered unclear or bad, it should be rewritten.
- If there is a need for inline comments, they should be simple and descriptive.

Example:

```
// Good:
velocity = acceleration * time;

// Bad:
v = a * t;   // calculate velocity
```

3.2 Doxygen

Doxygen[1] is a tool for generating formatted code documentation. The tool reads a folder of source code as input, and generates a document formatted as in readable markdown format including `HTML`, `PDF` and LaTeX. The generator interprets comments that have been formatted in a standard fashion, such that the compiler can convey the comments as explanation or definition of the relevant class or method.

We plan to integrate this commenting standard in all C++-code we produce, to generate documentation of our source code.

Classes and methods should be introduced by a comment block starting with two `*`'s:

```
/**
 *    ... text ...
 */
void myMethod();
```

Classes must include the author(s) and the date of writing. Both classes and methods must include a brief description of its own functionality, and that of any parameters or return value.

Variables should be explained by a single line comment prefixed with "`///<`":

```
float gravity;   ///< A constant downwards force
```

The syntax for commenting is similar to Javadoc[2], containing various tags prefixed with "`@`":

`@author` is the author of a class or interface, repeatable if authors exceed one.

`@date` is the author(s) of a class or interface.

`@brief` is a short one-line description of a method.

`@param` is the definition of any parameter for a method or constructor.

`@return` is the return value from a method.

`@see` is a reference to a cross-referenced class or method.

[1] https://www.doxygen.nl/index.html
[2] https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

Example of commenting:

```
/**
 * @brief   Represents a train.
 * @author  John Doe
 * @date    January 2022
 */
class Train {
    public:
        /**
         * @brief Check if the train has enough fuel.
         *
         * Checks if the train has enough fuel,
         * by comparing its amount of fuel to
         * the fuel needed.
         *
         * @param fuelNeeded The amount of fuel needed.
         *
         * @return True if the train has enough fuel, false if otherwise
         * @see FuelManager::fuelBurnRate
         */
        bool HasEnoughFuel(float fuelNeeded);
    private:
        float fuel;   ///< The current amount of fuel for this train
}
```

4 Code Formatting

- `nullptr` should be used instead of `NULL`.
- `auto` shouldn‘t be used, except for:
  - binding a lambda to a variable.
  - iterator variables, where iterator‘s type is verbose.
  - template code.
- Curly braces should be places before the line break.

```
// Use This
void MyMethod() {
    ...
}

// Not This
void MyMethod()
{
    ...
}
```

- `enum` classes should be used instead of namespaced enums.

```
// Old enum
UENUM()
namespace Thing
{
    enum Type
    {
        Thing1,
        Thing2
    };
}

// New enum
UENUM()
enum class Thing : uint8
{
    Thing1,
    Thing2
}
```

- Floating point literals should always have a radix point:

```
    // Good
    float scale = 0.5f;

    // Bad
    float scale = .5f;
```

- Control flow using nested if-statements should be implemented as guard clauses:

```
    // Good
    if(player == nullptr) return;
    if(!player->isAlive) return;

    player->Destroy();

    // Bad
    if(player != nullptr) {
        if(player->isAlive){
            player->Destroy();
        }
    }
```