GAME 3121 - GAME ENGINE

CODING STYLE

DIRECTORY ORGANIZATION

MINIMUM REQUIREMENTS



New Game Plus

Professor: Galal Hassan

Students:

* Brody Paterson Mulligan
* Ekin Karayalcin
* Ricardo Shimoda Nakasako

# Motivation

Writing code is almost an art - just like writing a book, creating a narrative on logic. On the other hand, few are the books which are created by multiple hands at the same time (if any), thus giving the story cohesion and a certain, unique and consistent style.

The goal of this document tis to uniformize the production of code as much as possible so that we can have coherent narratives in code when we need to use it in the future, either to fix bugs, to expando its functionality or even to trim the parts that are no longer necessary.

# Recommendations

## File Extensions

Do not use non-standard extensions. We will use:

* .h -> for Struct and Class definitions
* .cpp -> for functions and bodies of functions in classes
* .hlsl -> for shaders
* .xml -> for xml data (no other data format will be supported)

(This is an initial list and will be updated when needed)

## Commentary Header

All files will have the following header:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Class Name:

\* Description:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

So that we can at least know what was the original intention of whoever created the file.

## The #define in Headers

All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be <NAMESPACE>\_<CLASS>\_H\_.

#ifndef RENDERING\_MESHCOMPONENT\_H\_

#define RENDERING\_MESHCOMPONENT\_H\_

...

#endif // RENDERING\_MESHCOMPONENT\_H\_

## Forward Declarations and macros

Avoid using forward declarations where possible as well as macros. Instead, #include the headers you need.

## Names and Order of Includes

Include headers in the following order:

* Related header (i.e. the class whose members the cpp file is defining)
* C system headers,
* C++ standard library headers,
* other libraries' headers,
* your project's headers.

## Namespaces

Always place code in a namespace. Namespaces should have unique names based on the interface that it is implementing. We should not use using-directives (e.g. using namespace foo).

## Nonmember, Static Member, and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Do not use a class simply to group static functions. Static methods of a class should generally be closely related to instances of the class or the class's static data.

## Local Variables

Place a function's variables in the narrowest scope possible, as close as possible to their usage, and initialize variables in the declaration.

## Structs vs. Classes

Use a struct only for passive objects that carry data - such as components that implement the Interface and hold the data; from the Actor everything else is a class.

## Inheritance vs Composition

Composition is often more appropriate than inheritance. When using inheritance, make it public.

## Operator Overloading

Avoid operator overloading at all costs - when implementing this with templated classes eventually this might generate a geometric explosion of compiled code.

## Access Control

Make classes' data members private, unless they are constants. This simplifies reasoning about invariants, at the cost of some easy accessors (usually const) if necessary.

## Declaration Order

Group similar declarations together, placing public parts earlier.

A class definition should usually start with a public: section, followed by protected:, then private:. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declarations together, and generally prefer the following order: types (including typedef, using, and nested structs and classes), constants, factory functions, constructors, assignment operators, destructor, all other methods, data members.

## Functions

Write Short Functions: Prefer small and focused functions.

### Reference Arguments

All parameters passed by lvalue reference must be labeled const.

# Naming conventions

## Classes, Structs and Interfaces

Classes will always be defined in Pascal Case. (ie.:CanvasRenderingComponent).

Interfaces will use the same pattern as classes - adding an I preceding its name (ie.:ICanvasRenderingComponent)

Functions will also be defined in Pascal Case.

## Variables

### Prefixes

All variables will be receive a different prefix differently according to their scope:

* Global Variables: - starts with a g\_ - ie.: g\_screenWidth
* Member Variables: starts with a m\_ - ie: m\_localParameter
* Parameters of constructor functions: starts with an underline (\_)
* Parameters of functions and local variables: nothing

### Casing

Camel case for the variable name - so that Parameter That Changes Stuff would be writtern like:

* Global: g\_parameterThatChangesStuff
* Member: m\_parameterThatChangesStuff
* Parameter of a constructor: \_parameterThatChangesStuff
* Local/Parameter of a function: parameterThatChangesStuff

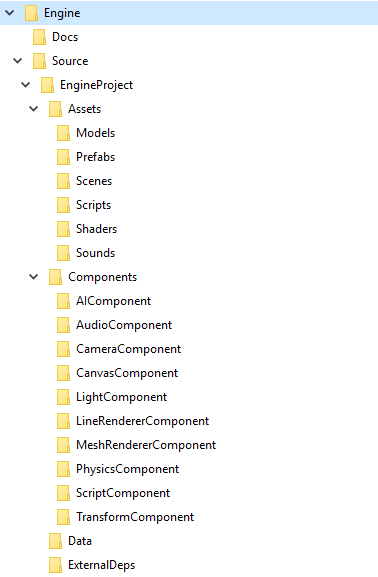
## Constants and Macros

Always in capital case - to give it the attention they need.

# Directories

Initially we'll have to directories: "Docs" for documents and "Source" for the Engine code.

For the C++ project we'll have the .sln file in the root directory and the csproj file one directory below - along with the file that contains the main class and executes the main loop of our code.



Based on this subdirectory, we'll have then, a subdirectory names "Component" which contains several subdirectories, one for each Interface implemented, a "Shaders" directory for the basic shaders supported by the Game Engine, a subdirectory named "Data" for all the basic xml files needed by the Engine, another Subdirectory named "ExternalDeps" for all the basic libraries upon which the project is dependent and, finally, a separate directory for Game Assets (images, scripts, 3D models, media files and additional shaders for the game itself)

# Minimum Requirements

* Processor: Core i3
* RAM: At least 4GB
* Operating System: Windows 10
* Hard disk: 5GB
* Graphics: At least an Intel Integrated HD Graphics 520