RISING TIDE's 2018 Style Guide

Created By The Rising Tide Team

Last Updated on December 5th, 2018

This document is incomplete partially due to most other style guideslines are enforced by the custom clang format file in the project.

Table of Contents

Comments

File Header Comments Style

```
/******************************************************************************/
/*!
  @file    [file_name.hpp]
  @author  [Author Name]
  @par     project: GAM200 - Tide Engine 2018
  @brief
    A small description or something.
    I don't think it's necessary to be too detailed just a general overview.

    Copyright (C) 2018 DigiPen (USA) Corporation.
*/
/******************************************************************************/
```

General Comment Style

- C++ or C style comments are both accepted
  - You should prefix notes with "NOTE(YourName):"
    - EX:
      - `// NOTE(Shareef): This is a C++ style comment.`
        `/* NOTE(Shareef): This is a C style comment. */`
  - And TODOs should contain the name of the person who's job it is to get done.
    - EX:
      - `// TODO(Amir): Get this shit fixed.`

Include Comments

- You should comment your includes for what you are using the include for, this helps document what items is being used from what modules.

File Naming

- C++ header and source files must have the '.hpp' / '.cpp' extensions respectively.
- C header and source files must have the '.h' / '.c' extensions respectively.
- All files must be in *lower_snake_case*:
  - All Engine code must be prefixed with "tide_"
  - All Gameplay code must be prefixed with "pg_" with "pg" standing for Project Gemini.
  - EX: "tide_particle_system.hpp" / "tide_particle_system.cpp"
  - EX: "pg_player_controller.hpp" / "pg_player_controller.cpp"

Include Guards

- I prefer the use of include guards rather than '#pragma once' since include guards are strictly conforming to the C/++ standard.
- They should be in all capitals (Macro Naming) and match the name of the file.
- The ending '#endif' should have a comment with the name of the include guard macro.
- EX:

```
// Imagine this file is named "tide_engine.hpp"

#ifndef TIDE_ENGINE_HPP
#define TIDE_ENGINE_HPP

// ... file contents here ...

#endif /* TIDE_ENGINE_HPP */
```

Order of Includes / Include Style

- Headers local to the project should be included before standard headers.
  - Rational is that it is more important to know what other local modules are being used from a dependency perspective while the standard library is more of an implementation detail.
- Includes should use "header.hpp" for local includes and <header.hpp> for standard includes.
- EX:

```
#include "engine/graphics/tide_vertex.hpp" /* Vertex        */
#include <unordered_map>                   /* unordered_map */
```

Classes and Structs

Class vs Struct

- This is very much a sematic battle but structs for POD (Plan Old Data) and Classes for everthing else.
- Classes should have exclusively private / protected data while struct should have exclusively public fields.
- Struct optimally should have no methods but constructors (or other setup methods) are welcome as they ease initialization.

Inheritance

- Avoid if you can composition is usually better. You should only inherit from engine interfaces.

Interfaces

- Should start with the 'I' prefix.
- EX:

```
class IEcsSystem { ... };
```

Friend Classes

- Should be used very sparingly. There are little cases where we should need to use this.

Casting

- Modern C++ style casts are prefered but even I don't adhere to them as often as a should as they tend to be overly verbose and visually noisy.

Indentation

Tabs vs Spaces

- We use **SPACES** not tabs.
- Preferably 2 spaces since that's how Mead likes it and it's a compact and neat style.

Namespaces

- Code within namespaces should be indented:
- All Engine code should be in the 'tide' namespace.
- If you must expose an internal implementation detail in a header file then put that in a nested namespace named 'detail'.
  - Right:

```
  // my_class.hpp
namespace tide
{
  namespace detail
  {
    class InternalClass
    {
      protected:
        void internalImpl();
      ...
    };
  }

  class MyClass
  {
    public:
      MyClass() = default;
      ...
  };
}

  // my_class.cpp
namespace tide
{
  namespace detail
  {
    void InternalClass::internalImpl()
    {
```

```
      ...
    }
  }
}
```

- Wrong:

```
  // my_class.hpp
namespace tide
{
namespace detail
{
class InternalClass
{
  protected:
    void internalImpl();
  ...
};
}
class MyClass
{
  public:
    MyClass() = default;
    ...
};
}

  // my_class.cpp
namespace tide
{
namespace detail
{
void InternalClass::internalImpl()
{
  ...
}
}
}
```

Switch Statements

- Switch cases should be indented.
  - Prefer to also put indented braces since I find it easier to read that way.
  - EX:

```
  // The braces are optional unless you declare a variable inside of a case.
  // but just to be universal I just add them regardless.
switch (condition)
{
  case CONDITION_ONE:
    {
      ++i;
      break;
    }
```

```
    case CONDITION_TWO:
      {
        --i;
        break;
      }
    default:
      {
        i += 5;
        break;
      }
  }
```

Spacing

- **Do not** place spaces around unary operators.
- **Do** place spaces around binary and ternary operators.
- Place spaces around the colon in a range-based for loop.
- Do not place spaces before commas and semicolons.
- Place spaces between control statements and their parentheses.
- Do not place spaces between a function and its parentheses, or between a parenthesis and its content.
- When initializing an object, place a space before the leading brace as well as between the braces and their content.
- When variabled are declared or assigned next to each other the names and '=' sign should be lined up.
  - EX:

```
Vertex   vertices[]       = {...};
Material material         = { Color::RED };
int      another_variable = 7;
```

Line Breaks

- Each statement should get its own line.
- An else statement should go on the separate line from the preceding close brace of the if.
  - EX:

```
if (condition)
{
}
else
{
}
```

- An else if statement should be written as an if statement when the prior if concludes with a return statement.
  - EX:

```
// The correct style
bool func(float a, float b)
{
  if (a > b)
  {
    return true; // Since this is a return statement
                 // Execution will not continue anyway.
```

```
  }

  return false;
}

// Don't include that extraneous else block.
bool func(float a, float b)
{
  if (a > b)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Floating Point Literals

- Always append .0 and .0f to floating point literals.

Null, True, False, 0

- In C++, the null pointer value should be written as *nullptr*. In C, it should be written as *NULL*.
- Tests for true/false, null/non-null, and zero/non-zero should all be done without equality comparisons.

File Naming Conventions

- File names should be all lowercase with words separated by underscores:
  - Ex: "batch_renderer.hpp"
- File Extensions:
  - If you are writing C++ code use ".hpp" and ".cpp" file extensions
  - If you are writing C code use ".h" and ".c" file extensions
  - This is because for some Compilers you can make them interpret the different files extensions in different ways.
- • If you can put the code files in the actual directory in the SVN then add it to the project. This is so the project folder is not cluttered up with random ass files everywhere.

Free Function Naming Conventions

- "lowerCamelCase" like member functions.

Class / Struct Naming Conventions

- Classes and Structs should use the CamelCase naming convention.
  - C++ Example

```
// Example C++ Class
class CamelCaseClass
{
  ... fields and methods ...
};
```

- C Example

```
// Example C Struct
// The 'CStyleCamelCaseStruct_t' can be left off but will be
// there in my code,
// it's for easy C and C++ Interop.
typedef struct CStyleCamelCaseStruct_t
{
  ... fields and shit ...

} CStyleCamelCaseStruct;
```

Member Variable Naming Conventions (C++ Only)

- Should always start with 'm_'.

Method Naming Conventions (C++ ONLY)

- Methods should be in lower camel-case.

Overriding Methods

- When overriding a virtual method in a subclass drop the virtual and add the 'override' keyword to the end. This cleans up the function declaration and also makes sure the compiler enforces all is correct with the method.

Functions that Operate on Struct(s) (C ONLY)

- When using C you should try to emulate C++ conventions to make things easier for other programmers using C++ to interop with your code.
- You can do this by defining these types of functions for your type:

```
// To Initialize 'Stack' Allocated Objects
// Ctor stands for "Constructor"
void            <struct_name>Ctor(<struct_name>* self, argument...);

// To Destroy 'Stack' Allocated Objects
// Dtor stands for "Destructor"
void            <struct_name>Dtor(<struct_name>* self);

// To Initialize 'Heap' Allocated Structs
// Using 'new' emulating C++
// This function should just malloc / calloc the self pointer and then call
// the 'Constructor' method.
<struct_name>*  <struct_name>_new(arguments...);

// To Destroy 'Heap' Allocated Structs
// Using 'delete' emulating C++
// This function should just call the 'Destructor' method and then free
// the self pointer.
void            <struct_name>_delete(<struct_name>* self);
```

- The first parameter should almost always be a pointer to the object you are manipulating so by convention I use the word "self" (stolen from languages such as Lua, Swift, Python etc...) although it can be anything. Just make it make sense.

- Functions that operate on structs format:

  ```
  <return_type> <struct_name>_<verb>(<struct_name>* self, arguments...);
  ```

  - For Example:

    ```
    void CamelCase_update(CamelCase* const self, double deltaTime);
    ```

- By using my rules of only using verbs for function names we can get rid of the use of "get/Get" for getters.
  - Function that just get a piece of state should just return that object.
  - Also generally your getters should return a **CONSTANT** pointer to that object.

    ```
    BAD:  Animation*        GameObject_getAnimation(CamelCase* const self);
    GOOD: const Animation*  GameObject_animation(const CamelCase* const self);
    ```

- Functions that return state as a boolean should use being verbs suchs as "is" and "are".

  ```
  bool CamelCase_isDead(const CamelCase* const self);
  ```

Enums Naming Conventions

- Enums should follow the same naming conventions as structs / classes, but enum values should be in all capitals.
  - A prefix for each enum value should be used in C but not C++.
- The enum values should end in a "_MAX" format to denote the maximum value the enum has. This may be left off but is often useful for error checking and using it as a loop condition.
- Enums in C++ should be "enum class" since they are more type safe and allow you to define the underlying datatype.
  - C++ Example

```
  // NOTE(Shareef):
  //   C++ Example Enum Class:
  //     The Prefix in this case is uncessary since
  //     you must always use the scope resolution operator ("::")
  //     to access the value.
enum class TestEnumCpp
{
  FIRST_VALUE,
  ANOTHER_ONE,
  TEST_ENUM_MAX,
};

  // NOTE(Shareef):
  //   Even better is to define the underlying type if you know
  //   the maximum value of the enum.
  //   This will help save on space if done diligently.
enum class BetterTestEnumCpp : uint8_t
{
  FIRST_VALUE,

    // NOTE(Shareef) : MAX may be a bad name since MAX is
    //                 a macro defined by some headers.
```

```
  MAX_VALUE
};
```

- C Example

```
// NOTE(Shareef):
//   C Example Enum:
//   The 'TestEnumC_t' can be left off but will be there in my code.
//   The Prefix 'TE' just stands for TestEnum but you can prefix it anyway
//   you want. Just be consistent with all values.
typedef enum TestEnumC_t
{
  TE_FIRST_VALUE,
  TE_ANOTHER_ONE,
  TE_MAX,

} TestEnumC;
```

Aliases (Typedef and Using)

- In C++ there is a new 'using' keyword.
  - For proper C++ style you should prefer to use *using* rather than *typedef* although there is no technical difference between the two.
  - (Although) *using* does provide some extra features if you are a template god.

```
  // NOTE(Shareef): This is prefered.
using MyFunctionPtr = void (*)(int someArg);
  // NOTE(Shareef): You can see how this is a bit messier with no advantage.
typedef void (*MyFunctionPtr)(int someArg);
```

Macros

- Macros should be declared at the top of the file if in the header.
- UPPER_SNAKE_CASE should be used for naming.

```
  // NOTE(Shareef): This macros gives me the max of two values.
#define MAX(a, b) ((a) > (b)) ? (a) : (b)
```

- '#undef'ing the macro when you're done with it is a good idea.

```
  // NOTE(Shareef): This is so this macro doesn't clutter the global
  //   'namespace'
#undef MAX(a, b)
```

Misc Code Style Notes

- Avoid using global variables as much as possible, if you absolutely needs a global variable make sure to have is scopes as small as possible. Such as using the 'static' keyword in a '*.c' file.
  - If a true global is needed prefix it's name with "g_"
- I prefer having curly bois ('{' and '}') on a new line but won't enforce it.