SFGE Technical Document

Elias Farhan

December 7, 2018

Contents

Introduction

This document is the reference for the coding style, that is a requirement for every commit on `develop` and `master` branches of the Simple and Fun Game Engine (SFGE).

It has been highly inspired by:

- SFML Coding Style
- Unreal Engine Coding Style
- Google C++ Style Guide

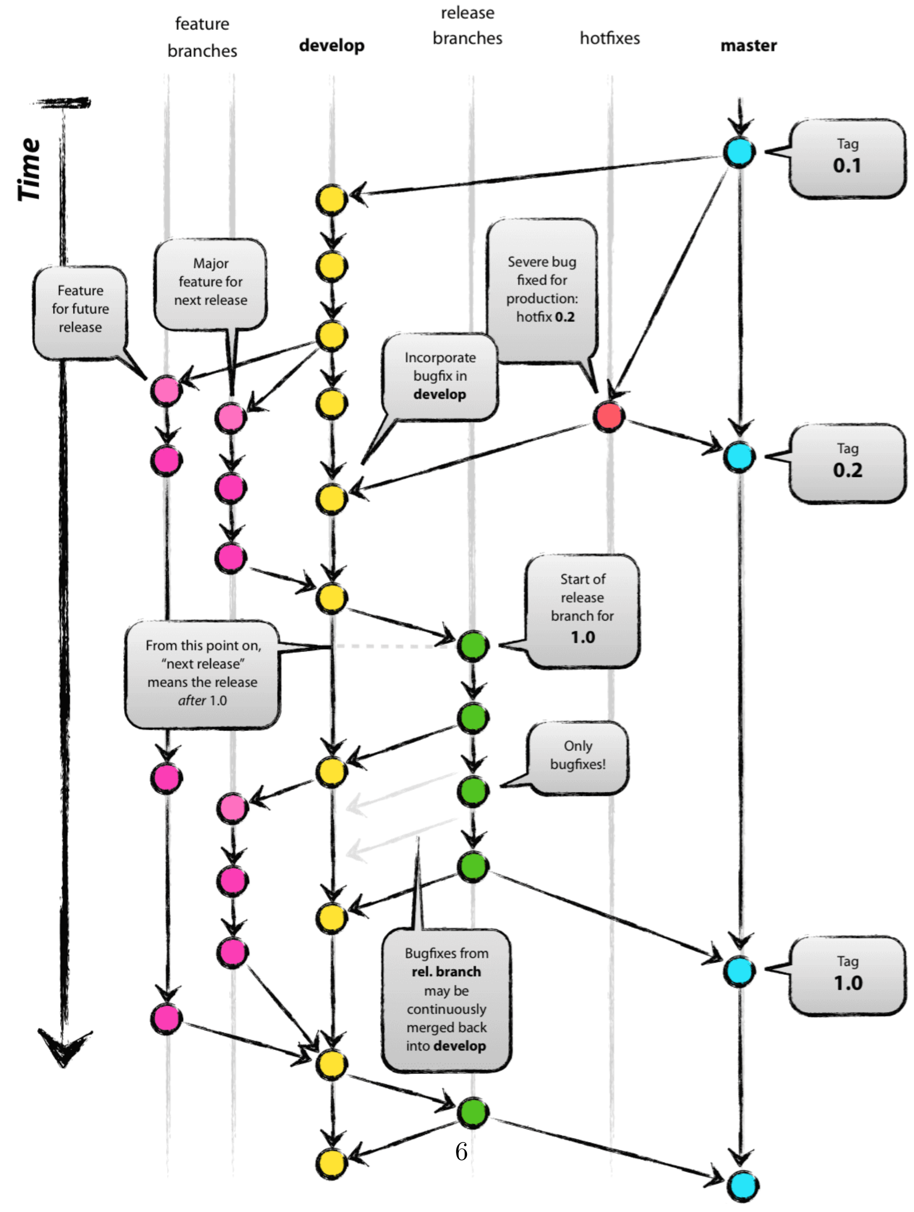
SFGE is distributed under the MIT License. Basically, you can do whatever you want as long as you include the original copyright and license notice in any copy of the software/source. You have to include at the beginning of every file of the project this comment:

```
/*
MIT License

Copyright (c) 2017 SAE Institute Switzerland AG

Permission is hereby granted, free of charge, to any
    person obtaining a copy of this software and
    associated documentation files (the "Software"), to
    deal in the Software without restriction, including
    without limitation the rights to use, copy, modify,
    merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom
    the Software is furnished to do so, subject to the
    following conditions:

The above copyright notice and this permission notice
    shall be included in all copies or substantial
    portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
     KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
    TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
    PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT
    SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
    ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
```

Chapter 1

Git

We are following the git model from Vincent Driessen [1].


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Chapter 2

Syntactical Conventions

2.1 Naming Convention

2.1.1 C++

Type	Convention
file name	prout_foo.cpp, prout_foo.h
type (struct, class, union, enum, typedef)	TitleCase
function, method	TitleCase
local, static and global variable	camelCase
private or protected member	m_TitleCase
enum constant, static const attribute	UPPER_CASE
JSON parameters	"path_prout"

2.1.2 Python

Type	Convention
file name	component_test.py
type (struct, class, union, enum, typedef)	ComponentTest
function, method	def foo_bar()
local, static and global variable	foo_bar
private or protected member	self.foo_bar
enum constant, static const attribute	UPPER_CASE

2.2 Indentation, Space, Parenthesis and Quarks

2.2.1 Declarations

`const` is placed before the type whenever possible. Reference & or pointer * are glued to the type (no extra space).

```
      T       obj;
const T       cobj;
      T&      ref;
const T&      cref;
      T*      ptr;
```

```
const T*        cptr;
      T* const  ptrc;
```

When a function or a method is not changing the content of an argument, it must be put as `const`.

```
/**
* \brief Print the values of an vector of int
* \param values The vector of int printed
*/
void PrintValues(const std::vector<int>& values)
{
        for(auto v& : values)
        {
                Log::Msg(v);
        }
}
```

2.2.2 Lines

1. There is only one instruction per line, except for readability in some switches.
2. Every definition (`class`, functions, ...) is followed by an empty line.
3. Braces are placed on new lines by themselves, except for `do ... while` loops.
4. `template` parameters and the rest of the function signature are on two different lines.
5. Every member constructed in the initializer `list` is on a new line.

   ```
   std::list<int> values
   {
           1,
           2,
           3,
           4
   };
   ```

6. If a line is too long it is intelligently broken up into a multi-line statement; e.g.:

   ```
   Color(Uint8(std::min(int(left.r) + right.r, 255)),
         Uint8(std::min(int(left.g) + right.g, 255)),
         Uint8(std::min(int(left.b) + right.b, 255)),
         Uint8(std::min(int(left.a) + right.a, 255)));
   ```

2.3 Parenthesis and Braces

2.3.1 Blocks

Blocks are always indented by one extra level, except for namespaces when there is only one used in the file.

2.3.2 `if/else/while` Statements

There are two forms of `if/else` statements: single-line or multi-line body. For an `if` statement that has only one instruction no braces are used. In any case a space always separates the keyword from the parenthesis. Every Brace is alone on the line – even if the while body is empty. E.g.:

```
if (audioContext) //Always put the brackets
{
    AlcDestroyContext(audioContext);
}
if (audioContext)
{
    // Set the context as the current one
    // (we'll only need one)
    AlcMakeContextCurrent(audioContext);
}
else
{
    err() << "Failed to create the audio context"
    << std::endl;
}
while ((nanosleep(&ti, &ti) == -1) && (errno == EINTR))
{
}
```

2.3.3 Logical Operators

If multiple && or || operators are used in the same boolean expression, then each part is guarded by parenthesis as soon as they consist of multiple sub-expressions themselves.

```
x < y                // no parenthesis
(x < y) && (y < z)   // with parenthesis
var && (x < y);      // variable not parenthesized
func() && (x < y);   // function call not parenthesized
```

2.4 Namespaces

The public API lives in the `sfge` namespace. The `sfge::priv` namespace is reserved for implementation details.

Anonymous namespaces are used when global variables are required, or for functions local to the current translation unit, in order to restrict their access to the translation unit.
No `using` directive should be used. Instead the full name is used everywhere.
Like written in subsection 2.3.1, `namespace` blocks are not indented, please change your settings in Visual Studio 2017: Tools→Options→Text Editor→C/C++ →Formatting→
Indentation: [ ] Indent namespace contents

```
//std dependencies
#include <map>
#include <string>
//Extern dependencies
#include <SFML/Window.hpp>
#include <imgui-SFML.h>
#include <imgui.h>
//Engine dependencies
#include <input/input.h>
#include <engine/log.h>



namespace sfge
{
```

Chapter 3

Python Binding

Introduction

3.1 Global

Most of the engine features are directly available from a python script. You can use the engine's systems statically (example: `sprite_manager` to access the SpriteManager of the Engine).
You can create new entities with a simple function call:

```
new_entity = entity_manager.create_entity(0)
#you can also choose your own entity number if it is
    available
```

To that new entity, you can add all kinds of components:

- Sprite:

  ```
  sprite_manager.create_component(entity, "data/sprites
      /prout.png")
  ```

- Body:

  ```
  body_manager.add_component(entity)
  ```

3.2 PyBehavior

PyBehavior are special kind of components attached to an entity and implemented in Python 3.6 and put into the scripts folder.
Here is an example:

```
from SFGE import *



class PlaySound(Component):
    snd_component: Sound
```

```
    def init(self):
        self.snd_component = self.get_component(Component
            .Sound)

    def update(self, dt):
        if input_manager.keyboard.is_key_down(
            KeyboardManager.Key.Space):
             self.snd_component.play()

    def on_trigger_enter(collider):
        self.snd_component.play()
```

Always import all from SFGE and inherit from Component. As you can see, several functions are available:

1. `void init()`: Called just after the scene loads. You can get all the references to other components (Sound, Sprite, PyBehavior, PySystem, ...) and initialize other variables.
2. `update(dt)`: Called each graphic frame and giving the delta time between the previous frame and the current one.
3. `fixed_update(dt)`: Called each physics frame and giving the fixedDeltaTime set in the Configuration.
4. `on(_trigger/_collision)(_enter/_exit)`: Called each time the collider of the entity is triggered/in collision with another collider.

You can access the entity number through the `self.entity` property. You can reference other components attached to the entity:

```
#reference an engine component
self.get_component(Component.Sprite)
#reference a pybehavior script called Prout
self.get_component(Prout)
```

You can then import the Behavior into the scene json, by adding into the scene component array of an entity with the script path like this:

```
{
        "name": "Prout Scene",
        "entities":
        [
                {
                        "components":
                        [
                        {
                                "type": 64,
                                "script_path", "scripts/
                                    prout_component.py"
                        }
                        ]
```

```
                    }
            ]
}
```

3.3 PySystem

PySystem are similar to PyBehavior except they are not attached to an entity, but to a scene, so no access to component and entity directly. You can override the following functions:

- `init()`: Called just after the scene loads. You can get all the references to other PySystem, create new entities, add components to them, etc...
- `update(dt)`: Called every graphics frame.
- `fixed_update(dt)`: Called every physics frame (dt value set in Configuration)

You can then import the System into the scene json, by adding into the scene system array an entry with the script path like this:

```
{
        "name": "Prout Scene",
        "systems":
        [
                {
                        "script_path": "scripts/
                            prout_system.py"
                }
        ]
}
```

3.4 PySystemCpp

PySystemCpp are PySystem implemented in C++. There are way more efficient than PySystem as they are compiled into assembly. You can implement them by creating a header file and a source file into the `extensions` folder and by adding an entry into the `extensions/src/python_extensions.cpp` file in the `void ExtendPython(py::module& m)` function, like this:

```
void ExtendPython(py::module& m)
{
        py::class_<ProutSystem, System> proutSystem(m, "
            ProutSystem");
        proutSystem
                .def(py::init<Engine&>());
...
```

You can then import the System into the scene json, by adding into the scene system array an entry with the class name like this:

```
{
            "name": "Prout␣Scene",
            "systems":
            [
                        {
                                    "systemClassName": "ProutSystem"
                        }
            ]
}
```

Chapter 4

Component Manager

Introduction

When thinking about engine component and to avoid adding bugs, there are several types of Component Manager, depending on the use of the components. What is important is the automation of the linking of the import process from the `SceneManager`, the resizing of the capacity of the ComponentManager called by `EntityManager` and the `DrawOnInspector` function called from the Editor. For each Component Manager, you have to implement (when it makes sense):

- an info type inheriting from `ComponentInfo` that implements `DrawOnInspector`.
- add an entry into the `ComponentType` enum.
- create a minimal Component class just keeping what it needs to be used by the ComponentManager to apply its feature (no path for Sprite, etc...).
- inherit the new Component Manager class from any class from the next sections.
- implement `CreateComponent` taking a json object into argument and filling the `m_Components` vector with the component at the right index.
- implement `AddComponent` to create an empty Component from Python.
- implement `DestroyComponent` that will be called from the `EntityManager` when the entity is destroyed.

Because Component Manager inherits from System, you can usse Init, Update and other specific functions directly.

4.1 Single Component Manager

Typical for sprite or rigidbody, it allocates one slot per entity and used the entity as an index.

4.2 Multiple Component Manager

Typical for script or collider, it allocates 4 times the number of entities (this value can be modified if needed). GetComponent will give the first component with the same entity.

4.3 Basic Component Manager

Custom class where you have to implement yourself the resizing mechanic.

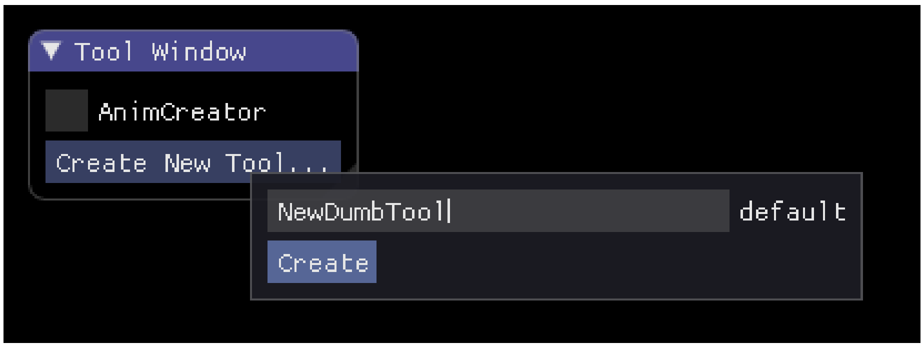
Chapter 5

Tool Programming

Introduction

The SFGE executable contains all the execution implemented in SFGE_COMMON as well as all the tools. To see the editor window, you have to press the key §.

5.1 Create a new tool



When opening the editor, you can click on the "Create New Tool..." button, write the new tool's name (in **Camel Case**, it's very important) and press "Create". It will create a new folder in the tools/ folder and create the header and source file that will be linked into the engine. You then have to quit the application, rerun CMake and then the tool will be available in the "Tool Window".

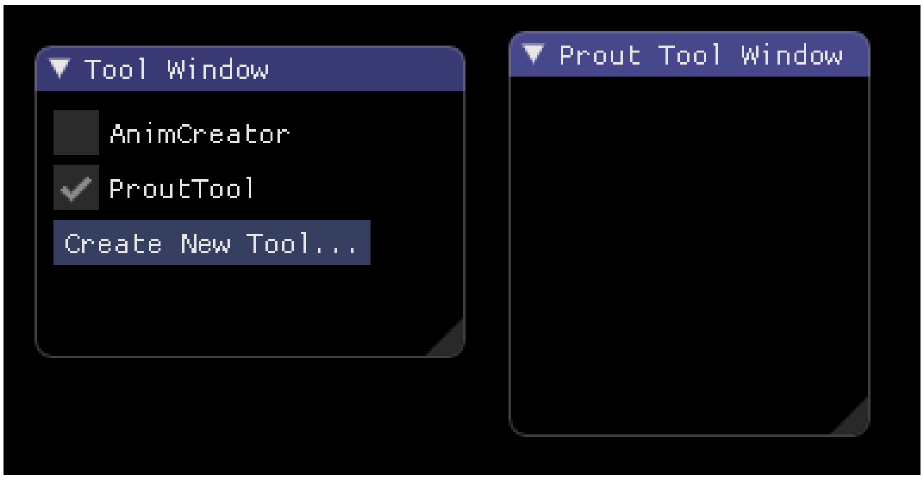
5.2 Implement your tool

In the the source file created by SFGE, you have three functions:

- `Init()`: used to initialized anything needed for the tool, get references from other systems.
- `Update(float dt)`: called every graphics frame, you can get special input if needed from the InputManager of the Engine, but you don't need to call any update on ImGui

- `Draw()`: called after all the tools Update, allow to separate the ImGui drawing calls from the Update, don't render ImGui, it is already being take care of by the Engine.

5.3 Open your tool



Simply check the checkbox of your tool in the "Tool Window" to open or close your tool.

Chapter 6

Documentation

6.1 Doxygen

SFGE uses Doxygen to generate the documentation in HTML or in LaTeX. Each `classe`, `struct`, `enum class`, function, method must be documented with a brief description, the parameters and the returned value for functions and methods.

The comment should be before the definition of the type with a comment starting with two stars and one star per line until the last line with a star and slash.

```
/**
* \brief The Texture Manager is the cache module of all
* the textures used for sprites or other objects
*
*/
class TextureManager : public Module<TextureManager>
{
public:
        /**
        * \brief load the texture from the disk or the
        * texture cache
        * \param filename The filename string of the
        * texture
        * \return The strictly positive texture id > 0,
        * if equals 0 then the texture was not loaded
        */
        unsigned int LoadTexture(std::string filename);
        /**
        * \brief unload the texture by removing a
        * reference count, if reference count is 0 then
            it
        * is unloaded from the cache
        * \param text_id The texture id stricLty positive
        *
        */
        void UnloadTexture(unsigned int text_id);
        /**
        * \brief Used after loading the texture in the
        * texture cache to get the pointer to the texture
        * \param text_id The texture id striclty positive
        * \return The pointer to the texture in memory
        */
    sf::Texture* GetTexture(unsigned int text_id);
private:

    std::map<std::string, unsigned int> nameIdsMap;
    std::map<unsigned int, sf::Texture> texturesMap;
        std::map<unsigned int, unsigned int> refCountMap;
        unsigned int increment_id = 0;
};
```

6.2 Guidelines

- Write self-documenting code:

```
// Bad:
t = s + l - b;

// Good:
totalLeaves = smallLeaves + largeLeaves
- smallAndLargeLeaves;
```

- Write useful comments:

```
// Bad:
// increment Leaves
++leaves;

// Good:
// we know there is another tea leaf
++leaves;
```

- Do not comment bad code - rewrite it:

```
// Bad:
// total number of leaves is sum of
// small and large leaves less the
// number of leaves that are both
t = s + l - b;

// Good:
totalLeaves = smallLeaves + largeLeaves
- smallAndLargeLeaves;
```

- Do not contradict the code:

```
// Bad:
// never increment Leaves!
++leaves;

// Good:
// we know there is another tea leaf
++leaves;
```

Chapter 7

Data Structures

7.1 Structures and Classes

`structs` are used to wrap up one or more variables together but do not use encapsulation; they are generally used by `classes` that do protect their members with protected or private modifiers. `structs` can not have constructors and should not have methods. They do not use access specifiers or inheritance.
In a `class`, the public interface comes first (usually with constructors at the top), followed by protected members and then private data. In a given access-modifier group static members are grouped together.

Chapter 8

Header

8.1 Includes

The inclusion order is as follows:

1. Standard library headers
2. Externals headers
3. SFGE headers

Example:

```
//STL includes
#include <list>
//Externals includes
#include <SFML/Graphics.hpp>
//SFGE includes
#include <engine/log.h>
```

8.2 Class Definitions

In a class, the public interface comes first (usually with constructors and special member functions at the top), followed by protected members and then private data. In a given access-modifier group, static members are grouped together.

```
////////////////////////////////////////////////////////////
//
// License text...
//
////////////////////////////////////////////////////////////

#ifndef SFGE_FILENAME_H
#define SFGE_FILENAME_H

//Externals includes
#include <...>
```

```
//STL includes
#include <...>
//SFGE includes
#include <...>
```

Chapter 9

Data

Textual datas are saved in JSON format using the Modern C++ JSON library. Always check user or JSON data input:

```
if (gameObjectJson.find("name") != gameObjectJson.end()
    && gameObjectJson["name"].type() == json::value_t::
    string)
{
     gameObject->name = gameObjectJson["name"].get<std::
         string>();
}
```

Bibliography

[1] **A successful Git branching model**, *Vincent Driessen*, Accessed [03.10.2017]. Link:
`http://nvie.com/posts/a-successful-git-branching-model/`

[2] **SFML Code Style Guide**, *Laurent Gomilla*, Accessed [03.10.2017]. Link:
`https://www.sfml-dev.org/style.php`

[3] **Unreal Coding Standard**, Epic Games, Accessed [03.10.2017]. Link:
`https://docs.unrealengine.com/latest/INT/Programming/Development/CodingStandard/`

[4] **Google C++ Style Guide**, Google, Accessed [03.10.2017]. Link:
`https://google.github.io/styleguide/cppguide.html`