**§ 1 – SFEngine Basics**

**§ 1.0.1 – SFEngine Basics**

SFEngine (Simple Fast Engine) is a 2D game engine designed from the core to be easy for game designers to make games without the usual large learning curve that accompanies large game engines.

**§ 1.1 – Threading Model**

SFEngine is multi-threaded, to allow complex or time-consuming work to be done without too negatively affecting the performance of the engine.

There are 3 main threads:

* Main Thread
* Render Thread
* Resources Thread

All assets (with a few exceptions) are loaded in a separate thread, inside which the resource manager resides.

**The Main Thread**

Inside the main thread all other threads are spawned, and where all other threads will join when shutting down.

The rendering window is created in the main thread, though rendering is not done from within the main thread to keep the rendering from stopping when the window is moved.

The main thread dispatches commands to all other threads, and as such should never be blocked.

The main thread is where a majority of the grunt work is done, though this may later be delegated to another thread.

The physics work is all done in the main thread, is addition to event polling. Physics code is made to be efficient so that excessive work is avoided.

**The Render Thread**

Inside the render thread is where all draw calls are made. Render calls are batched so that draw calls can be made in rapid succession.

The rendering thread can be set to only issue a display to the screen in set intervals to limit the framerate. This does not limit the time spent in other threads, though a global limiter can be used.

Rendering can only be done inside the rendering thread, as the OpenGL context is only valid in that thread. Making the RenderWindow valid in another thread is not recommended.

**The Resources Thread**

The engine uses a threaded resource management system. This thread does not work unless there are requests for resources that are outstanding. The thread will sleep and will be woken up by a small proxy that lives within the main thread and manages the storage of requests and the waking of the resource management thread.

The resource management thread manages the lifetime of all of the assets loaded, so one can guarantee that a resource will always exist while it is needed. A resource will not be destroyed until explicitly told to do so.

**§ 1.2 – The Memory Model**

The engine strives to avoid the unnecessary duplication of assets. All loaded assets are stored on the heap in a std::shared\_ptr object so that multiple objects can reference the same asset without having the asset be duplicated.

Manual memory management is not necessarily encouraged, unless the small overhead of a smart pointer must be avoided or you must have finely-tuned control of the memory, such as when manually packing.

The resource manager does not pool resources to be close together in memory. It uses the default C++ memory allocator to request the memory needed for an allocation. This can lead to heap fragmentation, so a custom memory pool may be implemented in the future. As of yet, no situation has arisen in which a memory allocated failed.

Factory methods exist currently to automatically create self-managing resources for developers:

* ActorFactory
* AnimationFactory
* LevelFactory (in which sub-factories are used)

Every factory uses the resource manager to allocate memory and create the object. The engine’s internals can handle a render call made to an asset that doesn’t yet exist to safeguard against assets taking a long time to load and being used before completely ready.

**§ 1.3 – The Event Model**

The engine’s events are polled in the main thread. As such, not blocking operation should be done in the main thread. The main thread issues commands to all other threads, so blocking it will block all other threads.

**§ 1.4 – The Rendering Model**

The engine’s rendering model is one where the delegation of determining which items to render is placed on the object that needs to be rendered.