technical desing document

SaNi Engine Core Systems

# Preface

## Version

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Description | Author(s) |
| 1.0 | 2.6.2015 | Initial version | NS & JN |
| 1.1 | 16.9.2015 | Cvar system | NS & JN |
|  |  |  |  |
|  |  |  |  |

## Version info

### Version 1.0

Contains initial documentation of the core system. Does not contain documentation for graphics, threading or profiling yet. These parts will be added later to this document.

## Purpose of this document

Purpose of this document is to document SaNi Engines core systems. SaNi Engine will be referenced as engine in this documentation. This document will cover every part of the core system layer.

## Use of this document

This document should be used as a reference by the developers of this project. It is good to refresh your memory once in a while, and every time the developer is uncertain of the technical structure of the system.

This document is intended to be read by software developers who are involved in developing the core system.

# Introduction

## Purpose

This document will cover every system inside the core system. These systems are:

* Primitives
* Logging
* Profiling
* Random numbers
* Assertions
* Unit testing
* Networking
* Localization
* Memory allocation and management
* Parsers
* Serialization/RTTI/Reflection
* Configuration
* Strings
* File I/O
* Math
* Threading
* Graphics wrapper
* Environment
* CVar system
* XML wrapper

Purpose of the core is to create basis for building the game engine. These systems are required to ease the development of the engine. Also a good base layer is a must for every working engine. These systems will be described in this document in detail.

Some parts of the platform independence layer will be developed during the development of the core system. This will contain mandatory systems such as file manipulation, primitives etc.

# Platform requirements and specifications

The system has to work on Windows (from version 7 till version 10), Windows Phone (starting from version 8), iOS (versions starting from 5.1), Android (starting from versions 4.0), PlayStation 4, Xbox One and Linux systems. The system will also work inside browsers, but this platform is not on high priority at the start.

For starters the core must work on Windows, Linux, Android and Windows Phone systems. Other platforms will be added in the future when it’s possible to test the system on them. The system will be developed so that it should be easy to add another supported platform.

Reason why older Android versions aren’t supported is the market share percentage of those versions. Versions below 4.0 only have around ten percent market share. Same goes with the iOS, versions older than 5.0 hold only around two percent market share. Windows Phone is another story. Windows Phone versions older than 8 have limited market share as well and Microsoft offers new updates and versions even to older phones. Another thing is that Windows Phone versions older than 8 only support DirectX9.

# Software requirements

Windows operating system is used for developing the system. The development environment will consist of Visual Studio 2013 Ultimate (student licensing). Compiler for Windows systems will be MSVC++12. Compiler for Linux systems will be Clang. Building for Android will require NDK (native development kit) and Tegra Studio. Building for Windows Phone systems requires Windows Phone SDK.

There will be five Visual Studio project files. These projects will target Android, Windows, Windows Phone, Xbox One and PlayStation 4. Premake4 will be used to generate gmake projects for Linux and XCode projects for Mac. Fate of web projects will be decided later and are not described in this document.

Version control will be hosted on Github.

Jenkins will be used as automatic building service. It will run all unit tests and compiles the system for all platforms. No idea about Xbox One or PlayStation 4 platforms yet, will figure this out later. There needs to be multiple build servers to able to build for all platforms.

## External dependencies

Only external dependency of the system is unit testing library named Catch.h. Doxygen will be used for documenting purposes.

## Language specifications

System and the engine will be written with C++ and using the C++11 standard. The core system does not contain parts that are written in other language than C++.

# System Overview

The system is a collection independent modules. The system will be compiled into a static library that the upper layers of the engine will use. Platform independence layer is the lowest most layer in the engine hierarchy.



The modules of this core system will be described in this part of the document. Most important functional and structural specifications will be documented. Also every module dependencies will be described. Calling convention will be from top to down. This means that no low level modules shall not depend from higher level modules. If this does not apply during the development process, the development will be paused and redesign is required.

## Overview

This part of document covers the two most low level layers that will be written by the developers.

## Namespaces

Listing of namespaces and their contents

**sani::debug:** Logging, Profiling, Assertions, Unit testing

**sani** (this is the root namespace)**:** Strings, Primitives, Memory management, Localization, Environment, Configuration

**sani::graphics:** Graphics wrapper, Graphics resources

**sani::math:** Math library and random numbers

**sani::io:** File I/O members, Parsers

**sani::thread:** Everything related to threading

**sani::net:** Everything related to networking

**sani::rtti:** Serialization/RTTI/Reflection

### Core systems

The core systems contains the most universal components. These components will be used across the engine.



### Platform independence layer

Platform independence layer will contain components that allow developing the engine regardless of the platform. These components are the lifeblood of the engine. Without these components, developing for multiple platforms painlessly is next to impossible. This layer if the engine will require best of the team’s design and development skills. If this layer is left with broken windows, all will fall. This layer contains next components:

* Graphics wrapper
* File I/O
* Threading
* Primitive types
* Networking layer

Networking and threading support won’t be added for some time. They are not discussed in this documentation. Reason why threading isn’t described yet is because few reasons:

* Developers don’t yet have vision how they want to use the interface
* Developers don’t have experience creating work or task systems
* Threading is not required at this part
* They don’t help with any problems at the moment

Even though both developers have used and studied threading in the past, it’s not that easy or clear how to develop your own threading/worker system. Threading will be added to this documentation later.

Work load of creating a networking layer is too high for the moment. Networking layer also requires threading to work in non-blocking way. Networking will be implemented later, when everything else that is required is implemented and working.

## System characteristics

System will consist from modules. Each module has its own unique purpose in the system. Modules should always have low coupling and high cohesion. No low level module should ever depend on high level module. Modules inside the system are allowed to communicate with each other, but goal is to keep this at minimum.

## Platform independence layer modules

As stated before, threading and networking are not described here yet. They will be added later.

### Graphics wrapper

Purpose of this module is to make rendering of the engine platform independence. Same interface works with all platforms and the user does not need to know or worry which rendering API is used. Module contains few classes and the main interface used to work with low level graphics.



Wrapper is used to handle context and window creations for different rendering API’s. Shaders and buffers will be wrapped as well. Any interface will not show any concrete implementation details about the wrapper.

Base class GraphicsDeviceBase defines the common interface to be used with wrappers and with the user interface GraphicsDevice. GraphicsDevices responsibility is to select the right renderer wrapper for current platform. This will be handled using C preprocessors.

There will be a common shader interface for all graphics interfaces. This interface will contain the common operations between each rendering API. The shader class is pure virtual class (interface) and does not contain any implementation details. There is also a platform independent way to bind graphical resources such as textures trough the graphics interface. These resources do not contain any implementation details, they are just interfaces.



There will be shader and texture implementations for each rendering API.

Graphics module will also contain window wrapper. Graphics device will contain instance of this wrapper. Most important role of the window wrapper is to store state and handle of the window. This handle is required to render the engine inside a WPF user control. Operations used to change the state of the window will be wrapped around the graphics interface, such as changing the resolution and applying these changes.

No interface design documentation exists yet. This will be created when the development of this module begins.

Classes of this module will be part of sani::graphics namespace.

### File I/O

File system implementation uses the same pattern and architecture as the graphics wrapper system. All implementations and the final interface share same, universal interface. The user should be able to traverse around the file system without needing to worry the underlying operating system. File system helps with few things:

* User does not need to worry about path separators (“/” and “\”)
* User does not need to worry about new lines or other special symbols
* Helps with permission checking
* Helps working with files and directories (modifying, reading)

While developing the file system module, dynamic memory allocation is prohibited.

Tentatively there will be File- and DirectoryInfo classes that will be used to work with files and directories across different platforms. Tentatively platforms will be detected using C preprocessors, each method has its own platform specific implementation, so no inheritance or polymorphism are required.

This subject need more research. Knowledge of platform specific file and directory operations are needed before planning and developing this module can start.

Classes of this module will be part of sani::io namespace.

### Primitive types

Strings and other primitive types such int, byte etc. will have their own type definitions.

## Core system modules

### Localization

Will not be included in the initial version of the engine. This module will documented in the future.

### Assertions

System will be using static assertions and runtime assertions that will be removed from the release build version. OpenGL

### Math

Defines mandatory math functions and classes needed in engine and game development. Will contain 2, 3, and 4 column vector implementations. These implementations use templates and they are intended to be used with int, float and double data types. There will be type definitions for vectors that use these types as template types. For example vec3f would create new 3 column vector and template type would be float. Module also contains quaternions, rectangles, points and matrices. These classes also use templates and will have type definitions.

### Logging

The logging system is service inside of engine. It will support logging to console, file, visual studio output window, network etc. The logging module has predefined variables to ease writing such as %file, %time etc. The module supports conditional logging with LOG\_IF. There can be multiple loggers which have unique logger id. It will also include timed logger.

### Strings

There is no point to create own string yet, as so many things can go south. This system will be using std::string as default which has its own type definition. This type definition should be used instead of std::string, so it can be changed later if we decide to create own implementation of string. As C++11 includes regex we don’t need to create any string utilities for it. If there needs to be any string utilities, they will be placed on StringUtils class.

### Serialization/RTTI/Reflection

There will be no reflection. There will be one interface which will be used for serialization and deserialization.

### Configuration

The configuration system of the engine is a mini configuration language built inside the engine. It’s similar to the source engines cvar system, and it is being called a cvar system inside the SaNi engine as well.

The cvar system consist of few important modules which are:

* Loaders
* Lexical analyzer
* Parsing and compilation
* The runtime cvar system
* “Auto exec” system

As stated before, the language itself is a kind of a mini language. The language contains few keywords, really simple syntax and strict syntactic rules. Visit the wiki page to find more information about the language.

The compilation steps are the following:

1. Locate the file containing the order in which the files should be “compiled”
2. Load files specified in this set first and then load other files that are found within the root directory
3. Process each files contents, execute least strict lexical analysis and generate tokens
4. Process each token by its type and generate intermediate representation of the language, do some more strict and specific analyzing here
5. Emit cvars
6. Construct the system

The configurator uses predefined values for default, but if configuration file contains values with same, the predefined values will be overwritten. Everytime the engine will be started, the configuration will be read and loaded to the engine. These configuration values will be used configuring for example graphics etc. Each time the engine is closed, all configuration data will be saved to the file. There can be multiple configuration files. These files will be read to same configuration manager. If there are duplicate values, the user will be warned and logged.

The system supports in-game console for configuration, but visual debugging tools will be created and documented later.

A cvar system will be implemented containing atleast the following things:

* Cvar parser
* Cvar container

There can be many files containing cvars and they all will be linked into one container. Some cvars are hard coded into the motor and can only be accessed inside the motor itself. The will be a predefined order that defines in which order the configuration files are loaded. The order is the following:

1. Load engine configurations
2. Load user defined configurations

Engine.cfg file will be looked first and parsed if found. AutoExec.cfg will be looked after the Engine.cfg has been found. When user does runtime changes to the variables, changes are not emitted to the files until the engine is closing. Some of the values will be hardcoded inside the engine, such as client and cheat variables. All configuration files are located in a folder that can be found at the root of the engine called “configuration”.

Syntax of the configuration language has simple rules:

* Every declaration must be on one line
* There can’t be undefined declarations
* Re-declarations with new values override the old value
* Types are resolved from the value declarations

The language contains few keywords which are:

* include that will the copy contents of given file to the call location of this keyword
* require that takes conditional statement inside, variables inside require can’t be accessed if the condition returns false
* exec that can be used to call functions

All functions that the exec supports are hardcoded inside the engine and can’t be user configured. We decided to limit functions to hardcoded ones, because it would get too messy and complex to give the user access to this part of the system.

Check wiki page for more information about the language itself.

### Memory management

There will be some basic allocators and smart pointers to be used with unmanaged memory. Allocators such as static allocator, single/double stack allocators etc will be implemented. There will be a custom reference counting pointer (smart pointer).

### Profiling

There will be a cross platform profiler for profiling. This part of the system will be documented and designed later.

### Random numbers

Will be part of the math module.

### Parsers

Will contain Json and XML parsers.

More research needed.

### Unit testing

Every module has their own unit test files. These files should be placed to tests folder. The system will be using Catch unit testing library. Tests should be broad and test most of the cases. Every test should pass before anyone can commit their files to version control, unless if it’s in another branch.

## Relations

Central class of the system will be engine. Engine is responsible for managing the services it contains. Services can be such things as the platform independent rendering service, which contains all the necessary things needed to perform simple rendering or the scene manager used for handling and managing game scenes (states). Services can communicate with each other. Example about communication between services is when rendering service might use the logging service for logging useful data, such as errors or information about its state during initialization.

# System Design

## Design methods and standards

System will be built using OOP principles. Inheritance and unnecessary coupling are not tolerated. While planning about creating a new base class or new inheritance hierarchy, please think twice before starting. Global variables and methods are not allowed, this also means that no singleton objects are allowed inside the system. Global methods and variables are allowed if they only have one state, meaning that utility and helper classes are allowed.

Every constant, method, and class will have namespace. Nothing will be put to the global namespace.

## Documentation standards

Documentation will be generated using Doxygen. While documenting, please use your brain. Don’t comment on stupid things. Example of a stupid comments:

int i = 0; // Set i to zero.

int add(int a, int b); // Add two numbers together. a = first param, b = second param.

Example of good documenting.

void flushBuffer(); // Flush and reset state of the buffer. Current data of the buffer will be lost.

Documenting of the interface will be contained in the header file. Logic documentation will be contained in the source file. Documenting will follow C# styled documenting and commenting. Example:

/// Flush and reset state of the buffer. Current data of the buffer will be lost.   
void flushBuffer();

Parameters should be documented if they aren’t documented by them self.

Self documenting code is not required to be documented.

Also, there should be an example about each class, how they can be used and when.

[Doxygen reference](http://www.stack.nl/~dimitri/doxygen/manual/commands.html)

## Naming conventions

### C++

Class naming  
- UpperCamelCase  
- Header and source files with lower\_case\_and\_underscore.extension  
- Avoid long names (AbstractBeanBeamFactoryFactoryAppletFactory)  
- Name should describe purpose of the class

Class variables and methods  
- Fields with lowerCamelCase  
- Methods with lowerCamelCase  
- Temp variables can have short names (such as i, j, k, res etc)

Constants and enums  
- UpperCamelCase  
- Enum fields with UpperCamelCase, enum names will user UpperCamelCase aswel

Methods  
- Name should describe what the method does  
- Parameter names are lowerCamelCase  
- Names are lowerCamelCase  
- Getter and setter methods start with get and set (getName(), setName(string value))

Namespaces  
- All lowercase (sani::io)

Defines/macros  
- WITH UPPER\_CASE

Type definitions  
- With UpperCamelCase

### C#

Only difference with C++ conventions is that methods and namespaces will be written with UpperCamelCase. Getter and setter methods are replaced by C# properties.

## Programming standards

### C++

Curly brackets   
- No new lines

If statements  
- One liners allowed if there will be no inner if statements (if (j < 10) doShit(j);)

Whitespace  
- Methods and operations should be grouped wisely using whitespace  
- Don’t cram everything next to each other, use whitespace to make readers happy

Identation  
- Tab identations  
- Whitespace when needed

Class initializer lists

Person::Person(string name, int age) : name(name),  
 age(age) {   
}

New line before “:” if the parameters list is too damn long.

Line length  
- Lines should not exceed ¾ of 1080p monitor with 100% zooming option.

### C#

Same rules apply with C# what are defined for C++ except every curly bracket should be placed on new line. Linq statements are separated by dots like so:

var elements = values.Where(v => v.num > 10)  
 .Select<EbinValue>(v => new EbinValue(v))  
 .ToList();