Software Requirements Specification (SRS)

by TophUwO

June 2024

Project **Noriko**

Component **Noriko**

Noriko is a cross-platform 2-D RPG game engine, focused on simplicity, ease of use, and scalability.

Table of Contents

Version History

The following table logs all changes made to this document during the analysis phase of the project. Author names may hold hyperlinks to the author's preferred way of contact. Dates are given in ISO-8601 **`MM-DD-YYY`** format. Version numbering is as follows: **`MAJOR.MINOR.PATCH-ITERATION`**. Version **`1.0.0`** marks the release version of this document according to which the software will be developed.

DATE	VERSION	CONTRIBUTORS	CHANGES
06-16-24	0.0.1-1	TophUwO	**INITIAL COMMIT** **ADD** table of contents
06-16-24	0.0.1-2	TophUwO	**ADD** version history **DO** accent color setup
06-16-24	0.0.1-3	TophUwO	**ADD** remaining major sections **FIX** formatting of tables
06-17-24	0.0.2-1	TophUwO	**ADD** initial content for all sub-sections within section *1. Introduction*
06-17-24	0.0.2-2	TophUwO	**ADD** text contents for section *2.1. Product Perspective and System Context*
06-17-24	0.0.2-3	TophUwO	**ADD** CTXD for Noriko's ecosystem **ADD** remaining text for section *2.1. Product Perspective and System Context*
06-17-24	0.0.2-4	TophUwO	**ADD** initial content for section *2.3. User Classes and Characteristics* **ADD** content for section *2.5. Design and Implementation Constraints*
06-18-24	0.0.3	TophUwO	**ADD** contents for section *2.6. Coding Style*
06-18-24	0.0.4	TophUwO	**ADD** contents for section *2.4. Operating Environment*
06-18-24	0.0.5	TophUwO	**ADD** product feature overview
06-18-24	0.1	TophUwO	**ADD** contents for sections *2.7. User Documentation* and *2.8. Assumptions and Dependencies*

1. Introduction

The following section provides basic information on not only the product described by this document, but also the document itself, such as document conventions, structure, and various pointers for readers.

1.1. Purpose

This document is a Software Requirements Specification (SRS) for the component called **`Noriko`** of project **`Noriko`**. Project Noriko is a cross-platform 2-D role-playing game (RPG) game engine. Other documents of this type exist for components **`NorikoEd`** and **`NorikoRt`**. Throughout the remainder of this document, the term **`Noriko`** will refer to the game engine component only.

1.2. Document Conventions

The document at its core closely adheres to IEEE 830: Software Requirements Specification. Additional sections providing miscellaneous information may exist, especially in the latter half of this document. The document's language is English (United States). Terms which are **`highlighted like this`** may be either technical terms, variables, or other information of special significance. In the Version History table, terms in **`CAPITAL BOLDFACE`** signify commit message actions, whereas terms in *`underlined highlighted italics`* are links to other sections within this document. When external links are inserted, a superscript `(»)` is appended to the link texts, whereas e-mail or other contact links are annotated via a superscript `(@)`.

1.3. Intended Audience and Reading Suggestions

The document is to be understood by all stakeholders. That includes project managers, designers, software engineers, and programmers. What sections are most important depends on the individual's function in the project. While project managers should be familiar with the entire document, software engineers and designers should focus on the latter half of this document which delves into low-level details of the system.

1.4. Product Scope

Games have seen a rapid increase in popularity in recent decades. With a milliard of genres to choose from, effortlessly crossing country-, demographic-, as well as

cultural boundaries, games have become a vital part of today's entertainment industry. Whereas today's focus of AAA studios is mostly on realistic 3-D worlds, there was a time not long ago when computers and especially consoles lacked the hardware power to simulate scenery this complex. In the 1990s and early 2000s, handheld consoles became increasingly prevalent, with the Nintendo® Game Boy® being introduced as far back as 1989. Due to the stringent hardware limitations of early-generation handheld devices, games had to not only heavily optimize game- and drawing routines, but also had to place strict requirements on complexity as well as code size.

Having grown up in the early 2000s, I came in contact with handheld consoles, with the Nintendo® Game Boy Color® being my first one. I quickly developed a deep interest in RPG games which were virtually always set in tile-based worlds due to the aforementioned limitations. This project, Noriko, aims to recapture the iconic charm that managed and still manages to captivate millions of kids across the globe by focusing on what made these games special, with the objective of making development of such games as easy, accessible, efficient, and scalable as possible.

1.5. References

The following table contains resources (web pages, books, articles, ...) to key topics and technologies mentioned throughout this document. They may serve as an introductory pointer to the reader; however, sticking to them is not required. Additional research is advised.

TERM	TYPE	DATE	INFO

1.6. Applications and Tools Used

This section lists all tools and external applications used for creating this document, grouped by scope.

SCOPE	TOOL(S) USED

LAYOUT & TYPESETTING	Microsoft® Word® 365 (Version: 2405)
OOA DIAGRAMS	Eclipse® Papyrus (2023-12 (4.30))
CODE SNIPPETS	Microsoft® Visual Studio® 2022 Community (Version 17.8.4)

2. General Description

This section focuses on key aspects of the system, such as high-level features, user-, and system context analysis.

2.1. Product Perspective and System Context

All three main components of Noriko live in the same ecosystem and are designed to depend on one another in order to function properly. The ecosystem is largely self-sufficient; dependencies from outside the ecosystem are non-existent or minimal.

The following system context diagram visualizes the relationships between Noriko's three main components. Optional and external components are marked as such. Optional components may extend the system's functionality but are not required to exist for the system to provide all necessary features.

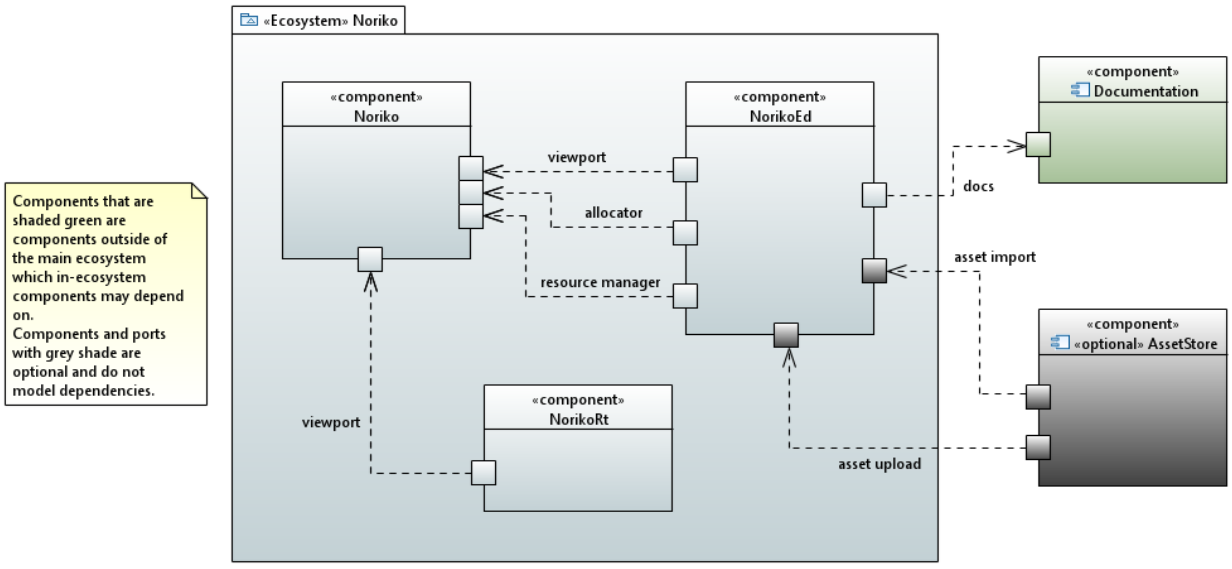

Figure 1: system context diagram (CTXD) showing Noriko's main three components as well as major dependencies.

This document serves as the requirements specification for the game engine component of Noriko. The components **`NorikoEd`** and **`NorikoRt`** are explained in detail in other documents.

2.2. Product Functions

Noriko is a 2-D RPG game engine ecosystem, featuring an engine component with the same name, **`NorikoEd`**, the in-ecosystem editor implementing the content creation pipeline, and **`NorikoRt`**, the runtime directly instantiating the engine component. In the following, only the engine component's core features are outlined:

- render 2-dimensional tile-based worlds and dynamic entities
- tiles can contain interactive event data
- support for animations
- support for advanced image effects (alpha blending, …)
- multiple UI overlays
- customizable GUI
- extensible debugging features (also graphical)
- scripting for events, dialogue, and entities
- multiple rendering technologies (APIs) supported
- particle system
- shader support
- networking
- audio support

2.3. User Classes and Characteristics

Noriko's does not feature a user hierarchy. The user is the developer or player of the game that was created using Noriko's tools.

2.4. Operating Environment

Noriko is intended to be run on desktop as well as embedded platforms. There are generally no minimum system requirements, except that the platform needs to have a graphic output raster device (i.e., a display). The only requirement is that a C17-conforming C-standard library is available on the target platform. Features that are infeasible to implement on less powerful platforms are optional. What features are infeasible must be determined during the implementation and testing phase.

For desktop and mobile platforms, here are the minimum system requirements that must support all product features outlined in this document:

PLATFORM	MINIMUM OPERATING SYSTEM VERSION
MICROSOFT WINDOWS	Microsoft Windows® 7 (Build: 7601)
LINUX	Linux® Kernel Version 3.2
MACOS	Apple macOS® 10.10.0 'Yosemite'
NINTENDO SWITCH	Nintendo® Switch® Firmware v16.0.0

Support for platforms not listed in the table above is optional.

2.5. Design and Implementation Constraints

The game engine is to be implemented in ISO C17 (ISO/IEC 9899:2018) where possible. Using APIs (application programming interfaces) which are written and exposed to the user via non-C interfaces is permitted but appropriate wrappers must be provided. There is no limitation as to what external libraries are allowed to be used, as long as they are released under a permissive software license such as the *BSD 3-clause license*(») or the *Apache 2.0 license*(»). As for the source code and inline documentation, the required language is **English (United States)**.

2.6. Coding Style

This section describes the coding style used in Noriko's codebase. This coding style is to ensure consistent quality, optimized for easy-to-read code and a variety of display hardware.

2.6.1. Rules

The following table categorizes language features into **elements** and lists the style rules that are valid for the given element.

ELEMENT	RULE
FILE NAMES	○ not longer than 24 characters ○ all lower-case ○ contain only alphanumeric characters and _ 'underscore'
GLOBAL FUNCTION NAMES	○ not longer than 64 characters ○ start with **Nk** prefix ○ use camel-case
GLOBAL IDENTIFIERS	○ not longer than 32 characters ○ prefixed with **gl_** ○ use camel-case
GLOBAL CONSTANTS	○ see row **GLOBAL IDENTIFIERS** ○ infix **_c_** added between **gl_** and the variable name
STRUCTURE NAMES AND TAGS	○ see row **GLOBAL FUNCTION NAMES**

VARIABLES DECLARED IN STRUCTS	○ not longer than 16 characters ○ prefixed with **`m_`** (normal) or **`mp_`** (pointers) ○ use camel-case
FUNCTION POINTERS DECLARED IN STRUCTS	○ not longer than 16 characters ○ use camel-case
LOCAL VARIABLES	○ see row **`FUNCTION POINTERS DECLARED IN STRUCTS`**
LABELS	○ not longer than 16 characters ○ all upper-case ○ prefixed with **`lbl_`**
ENUM CONSTANTS	○ use PascalCase for identifier ○ prefix with user-defined scope identifier, followed by a **`_`** 'underscore', indicating the enum the constant belongs to ○ use PascalCase for prefix as well
MACROS	○ all upper-case ○ use **`NK_`** prefix

2.6.2. Formatting

The following formatting rules apply:

1) single lines not longer than 120 characters
2) opening **`{`** 'brace' are placed on the same line as the identifier or control block
3) control blocks with one statement do not use **`{}`** 'braces'
4) indirection **`*`** 'asterisk' attach to the identifier instead of the type
5) **`else (if)`** are placed on the same line as the closing **`}`** 'brace' or on the line after the statement if **`}`** are omitted due to rule **`3`**
6) use 4 (four) spaces for tab-stops (NEVER use actual tabs)
7) nested preprocessor directives are indented just like code
8) add one space after control blocks and between control block expression and opening **`{`** 'brace'
9) do not use parenthesis for **`return`**, **`sizeof`** and alike (unless syntactically and/or semantically required)
10) insert space after commas in argument and initializer lists
11) always use **`/* */`**-style comment blocks
12) use any valid DOXYGEN-style comments for inline documentation
13) add spaces between (binary) mathematical operators and their operands
14) macro values are always enclosed in **`()`** 'parentheses'

15) in a declaration, annotations come first, then storage classes, then the type, and then type qualifiers
16) grouped declarations that span multiple lines are aligned so that their identifiers start on the same column
17) labels are always placed one level of indentation to the left of the current level in their own separate line
18) use blank lines liberally – within function, group code based on what it contributes to
19) in implementation (*.c) files, static functions come before exported functions
20) symbol order within header and implementation files:
    a) `#include`
    b) `#define`
    c) `struct`/`enum`/`typedef` declarations & definitions
    d) `static` functions
    e) exported functions

2.6.3. Full Example

The following code showcases the coding style described in the above sections. Lines may have a comment such as **`/* 8. (& ...) */`**, indicating what formatting rules were applied.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>



/* Demonstrating nested preprocessor directives. */        /* 11. */
#if (defined NK_LEVEL_1_PREPROC)
    #if (defined NK_LEVEL_2_PREPROC)                       /* 7. */
        #define NK_LEVEL (3)                               /* 14. */

        #include <windows.h>
    #elif (defined _DEBUG)
        #error Cannot compile in debug mode.
    #endif
#endif
#define NK_ANNOTATION


/* Demonstrating global variables. */
NK_ANNOTATION static int const gl_c_displayWidth = 1080;   /* 15. */
```

```c
/* Demonstrating types. */
typedef int NkErrorCode;

/**
 * \brief This is a DOXYGEN-style comment.
 *
 * This is the detailed description for this declaration.
 *
 * \note  This is a note.
 */
typedef struct NkStaticApplicationContext {                 /* 2. */
    uint32_t  m_uintVal;                                    /* 16. */
    uint32_t *mp_uintPtrVal;                                /* 4. & 16. */

    void (__cdecl *getObject)(char const *const id);        /* 4. & 15. */
};
/* Demonstrating enumerations. */
typedef enum NkErrorCode {                                  /* 2. */
    NkErr_Ok,
    NkErr_Unknown,

    NkErr_NotImplemented,
    NkErr_OutOfMemory
} NkErrorCode;


/* Demonstrating functions. */
NK_ANNOTATION inline NkErrorCode NkCreateWindow(            /* 1. & 15. */
    uint32_t width,
    uint32_t height,
    uint8_t bbp
) {
    /* Validate parameters. */
    if (!width || !height)                                  /* 3. & 8. */
        return NkErr_NotImplemented;

    /* Do stuff. */
    void *ptr = calloc(1, sizeof width * 32);               /* 4. & 9. & 10. & 13. */
    if (!ptr)
        return NkErr_OutOfMemory;
    else if (ptr == 0x01) {                                 /* 5. */
        printf("Pointer was 0x01.\n");

        goto lbl_ERR;                                       /* 9. */
    }

    /* Clean up and return. */
    free(ptr);                                              /* 18. */
    return NkErr_Ok;

    /* Clean up in case of error. */
lbl_ERR:                                                    /* 17. */
    free(ptr);
    return NkErr_Unknown;
}
```

2.7. User Documentation

Since the engine itself is not used by players directly, its user documentation is to be provided in the form of inline documentation, generated with the help of **DOXYGEN**, and tutorials.

2.8. Assumptions and Dependencies

For the engine to function properly, a readable and writable filesystem must be present on the side of the target platform as well as a way to manually manage memory. Furthermore, the engine depends on the C-standard library for various core functions, so it either must be present on the host platform or deployed alongside the engine.

3. External Interface Requirements

4. System Features

5. Non-functional Requirements

6. Other Requirements

7. Optional Requirements

8. Future Requirements and Plans

Appendix A: Glossary

Appendix B: Supplementary Resources

Appendix C: Original User Story

Appendix D: Analysis Models

Appendix E: TBD