SOFTWARE REQUIREMENTS SPECIFICATION

for

GET
2d Game Engine Toolbox

Version 1.0 approved

Prepared by Kier Lindsay and Léo Abiguime

CMPT-376W

April 9, 2020

Contents

Revision History

Name	Date	Reason For Changes	Version
Kier Lindsay	April 9, 2020	Inital Copy	1

1 Introduction

1.1 Purpose

This document refers to the GET software, version 1.0. GET stands for Game Engine Toolbox and it is a two-dimensional (2D) game engine designed to bridge a gap between existing graphics library and large complete three-dimensional (3D) game engines. It will extend the capabilities of graphics library and provide common systems and feature needed to develop a game without forcing the user into a large complex engine. In its first iteration (version 1.0), GET is based on the C++ SFML engine. This SRS will cover the entirety of GET

1.2 Document Conventions

This document follows the IEEE SRS template. It also makes use of the KISS mentality in order to make things easy to parse.

1.3 Intended Audience and Reading Suggestions

This document is intended for the developers who will be implementing the game engine. We would also like to consider users of the engine as this document may offer a high level overview of what the product offers and its design philosophy. We recommend that developers start by reading chapter 2 (Overall Description), especially section 2.2 (Product Functions) as they will be in charge of implementing the components mentioned there. Then, they can move on to chapter 4 which elaborates the system features of each module of the software.

1.4 Project Scope

This project is meant to complement an existing graphics library providing the core functionalists all games require and tools for developers to use. It is meant to provide a more advanced and highly customization starting point for game developers who want to have full control over there game. It is also meant to be modular so that developers are not forced into using our designs and have the freedom to incorporate as many or as few of the features we provide. The current iteration of GET focuses on SFML which is a C++ multimedia library. It is used as the backbone of the software and the aim is to cover more libraries in the next versions.

1.5 References

SFML IMGUI SDL2 RayLib

Minimal programing guides.

others as we write them or references to documentation for implementation

<List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

2 Overall Description

2.1 Product Perspective

The product being specified in this SRS is completely new. It advertises itself as a game engine that comes handy when using the SFML library for the reason that it augments the functionalities of that library. It originates from the need to develop games faster without compromising on stability and security. It is self-contained in the sense that it does not have any dependencies that require additional steps to be taken by its user, even though it relies on the SFML library. However, since it comes as an addon to the aforementioned library, GET can only run on an SFML dependent application.

2.2 Product Functions

GET is organized into three modules. Each module provides a set of functions that are enumerated below:

1. Engine Module
    - Timing - Setups at maintian a core game loop
        - Allows users to choose between using real deltaTime or a fixed tick time so that each tick is a consistent time
        - Allows users to set tickrate and framerate independently
        - Allows users to hook into update and render calls
    - States - Manages switching between game states
        - Allows users to bind a number of game states to an engine, each with its own update and render functions
        - Allows users to toggle states
    - Assets - Manages the loading and management of game assets such as fonts and textures.
        - Supports Fonts
        - Supports Sprites
        - Supports Textures
        - Supports Animations
        - Supports Sound files

2. Physics Module
    - Body - Links the physics module to a shape in order to apply constraints to it through the manager or functions
    - Manager - Used to hold bodies and can be linked to a background of the UI module
    - Functions - A set of functions that can be applied onto bodies or the managers
3. User Interface
    - Shapes - Create complex shapes
        - Allows users to generate a shape using a sprite file loaded as an asset
        - Allows users to generate humanoid shape based on a set of parameters
        - Allows users to link shapes to a body
    - Background - Link a set of shapes together to form an object on top of which complex shapes can be displayed
        - Offers a set of parameters to initialize a background
        - Allows users to initialize the background object with non-complex shapes as a parameter
        - Allows users to link the background to an instance of Manager class from the physics module
        - Allows users to add music to the background

2.3 User Classes and Characteristics

We can distinguish between two categories of users: full users and partial users.

Full users will use this engine as their starting point and build there game around our components and tools. That means that they do not use GET to augment the functionalities provided by SFML but rather as a full fledged game engine.

Then, we have **partial users** who will SFML as their starting point and may incorporate some components of our software into there project.

2.4 Operating Environment

Since our software is dependent on SFML, we should expect a similar operating environment. As such, we have the following:

- Operating System: Windows(10, 8, 7, Vista, XP), Linux, macOS. x64 and x86 systems are both supported.

- Dependencies: SFML, OpenGL, Imgui
- Language Target: C++

2.5 Design and Implementation Constraints

This software is not intended to be used to develop 3D projects because SFML does not inherently support 3D. Therefore the software architecture should be designed in a way that makes it hard for users to develop 3D applications. Moreover, GET is not meant to be supported in any programming language other than C++ since partial users, as defined in the user classes, might use SFML as the backbone of their application and SFML is only supported in C++.

2.6 User Documentation

The following user documentation components will be delivered along with the software:

- README file: will be included in the repository from which GET will be downloaded. It will serve as a setup guide.
- Tutorials: will be available on the official website to get the user started with the product.
- API Reference: will elaborate on each module to provide users with the expected behaviour and the intended use of each component.

2.7 Assumptions and Dependencies

It is assumed that the user's system is running of SFML version 2.2 or above. Any version below that is seriously outdated and might lead to unexpected behaviors.

3 External Interface Requirements

3.1 User Interfaces

The user interface will be the user's terminal. GET will interact with the user interface through log messages. We can distinguish between the following types:

- Information: Used to display basic events. Ex: user running the software.
- Status: Used to display 'significant' events that require a status code.
- Warning: User or system errors that do not cause failure of a component.
- Error: User or system errors that cause failure of a component.

The following requirements must be respected when dealing with user interface's log message.

REQ-UI1: On succesfully running GET, a log message that includes the name and the version of the software must be printed out.

REQ-UI2: A log message must be printed out everytime an event occurs.

REQ-UI3: Each log out message must indicate its category.

REQ-UI4: All logs must be saved inside a log.txt file under the main folder.

3.2 Software Interfaces

As stated numerous times, this game engine is based on SFML. Therefore, the data must be stored in a way such that it can interact with SFML in the case were the users are only partial users.

REQ-SI1: Every class must implement an SFML class as a superclass.

REQ-SI2: Every SFML function must behave as intended on custom objects defined by our software.

REQ-SI3: Users are able to use SFML objects and GET objects interchangeably.

4 System Features

4.1 Engine Module

4.1.1 Description and Priority

This feature proves the core game loop and state functionality it is an essential feature and is high priority. The timing component we allow users to manage the main loop speed. and when it calls the current states update and render functions. These can be different so games can run at a constant physics rate and scale that as necessary but only render when needed or at a fixed frame rate. It will have game states that can be switched between from events within the update function. This is meant to be extended by the users game and the update and render functions overridden with games specific code.

4.1.2 Stimulus/Response Sequences

- Stimulus: Time adjusted
  Response: Internal time variables set so the new timing take effect the next tick

- Stimulus: State changed
  Response: the corresponding states update and draw function well be used on the next tick.

- Stimulus: New Tick
  Response: Current states update function is called and if it is time to draw a frame the render function is also called

- Stimulus: Camera Set
  Response: Current states Camera is set and the cameras transformation from world coordinates to Camera coordinates is applied while rendering

4.1.3 Functional Requirements

REQ-ET1: Tick rate and Frame rate must be controlled independently

REQ-ET2: When calling Update in a tick the Tick time must be sent as a parameter

REQ-ET3: If ticks cannot be computed in time render frames may be dropped.

REQ-ES1: The update and render function should be called for all active states.

REQ-ES2: States should be able toggle states from within the update function.

REQ-ES3: States Should have an Identifier, Update function, and Draw function

REQ-ES4: States should accept a camera transformation which applies to the render function.

REQ-ES5: States should be able to be given indexes for rendering order. This should default to the order they were added

4.2 Physics Module

4.2.1 Description and Priority

The Physics Module will be a high priority. This will have three components the *Physics Manager* which can be used to hold *Physics Bodies* and manage the relationships between them as well as the *Physics Functions* which will provided many useful physics calculations and can be used on bodies independently or be used by the physics manager to update the bodies it manages..

4.2.2 Stimulus/Response Sequences

- Stimulus: New Body Created
  Response: Body Initialized with provided parameters and is returned to user for use

- Stimulus: New Manager Created
  Response: Physics Manager Initialized with universal constants provided and environment for bodies is setup

- Stimulus: Body Added to manager
  Response: Body is added to the manager and will be updated and rendered all world relationships will be applied to it

- Stimulus: Relationship Added to manager
  Response: Bodies referenced by the relationship will have it applied to them

- Stimulus: Collision Detected between Bodies
  Response: Bodies are separated and appropriate forces are applied.

- Stimulus: Body Updated delta time
  Response: All applied relationships are computed and bodies parameters are updated

- Stimulus: Body Rendered
  Response: Body is drawn to screen at corresponding position

- Stimulus: Body Deleted
  Response: All relationships are removed and corresponding managers and bodies are notified then body is deleted.

4.2.3 Functional Requirements

REQ-PB1: The Physics Body class should store all physical properties required by the Functions this includes: Shape, Position, Velocity, Acceleration, Rotation,Rotational Velocity, Friction, Elasticity, Shape

REQ-PB2: The Physics Body class should have an *Update* function which updates the bodies property according to classical mechanics when called, this function will take in a delta time parameter

REQ-PB3: The Physics Body class should have an *Render* function which Draws the bodies shape. This should be designed to be overridden by users.

REQ-PB4: The Physics Body class should keep an list of relationships it is a part of so that is can apply them all when updated.

REQ-PF1: The Physics Function should accept universals constants as parameters or allow global defaults be set.

REQ-PF2: The Physics Function should include *Collision detection* between two bodies with *Minimum Translation Vectors* (MTV). It should include fast approximate bounding box collision detection as well.

REQ-PF3: The Physics Function should include *collision solving* functions which will apply the necessary forces to handle a collision between two bodies.

REQ-PF4: The Physics Function should include *gravity* functions to apply gravity between two bodies.

REQ-PF5: The Physics Function should include *constant force* functions to apply constant forces to a body these could be used to create world gravity or wind forces.

REQ-PM1: The Physics Manager should create an environment to hold Body' s as well as set world constants for the bodies it manages.

REQ-PM2: The Physics Manager should allow relationships to be made between bodies which define additional iterations. This includes forces and whether or not bodies should interact with each other.

REQ-PM3: The Physics Manager should have an *Update* function which applies all relations and updates all bodies

REQ-PM4: The Physics Manager should have an *Render* function which renders all bodies. This should allow for bodies to be drawn in a defined order

REQ-PM5: The Physics Manager should allow bodies to be grouped and relations or physics functions can be defined for the whole group or for each par of bodies in the group.

REQ-PM6: The Physics Manager should allow world Relationships to be set which apply to all bodies. These include global forces and n-body gravity as well as other optional relationships as appropriate.

REQ-PM7: The Physics Manager should allow preform collision detection between select objects this should be efficiently implemented so that full collision detection is only used when necessary not at all times.

4.3 User Interface Module

4.3.1 Description and Priority

The User Interface module provides the developer with functionalities that augment the graphics module of their game engine. It is meant to be built on top of the Physics Module as it is a visual interpretation of the constraints implemented there. Its priority level is low since it is not required for the game to run and its purpose is to enable the players to interact with the game.

4.3.2 Stimulus/Response Sequences

- Stimulus: New Humanoid Shape Created
  Response: Humanoid shape is initialized with its parameters and is returned to user for use.

- Stimulus: Humanoid Shape linked to Body
  Response: Humanoid shape is added to an already instantiated body from the Physics Module. Its behavior handling is passed to the Body's manager.

- Stimulus: New Miscellaneous Shape Created
  Response: Miscellaneous shape is created using the provided image files and is

returned to user for use.

- Stimulus: Miscellaneous Shape linked to Body
  Response: Miscellaneous shape is added to an already instantiated body from the Physics Module. Its behavior handling is passed to the Body's manager.

- Stimulus: New Background Created
  Response: Background is initialized with its parameters and is returned to user for use.

- Stimulus: Background linked to Physics Manager
  Response: Background is link to an instance of Physics Manager and all its property will be valid within the background.

4.3.3 Functional Requirements

REQ-SH1: The Humanoid and Miscelleaneous shapes are customs shapes that inherit all the attributes of an SFML shape.

REQ-SH2: The Humanoid shape is a custom class that adds the following attributes on top of the abstract class sf::Shape: head size, body size, body weight, height, weight, feet size, body color, hair color, hair type, eyes color, humanoid type.

REQ-SH3: The Miscellaneous shape is a custom class that uses convex shape to draw a loaded sprite that has been passed as a parameter.

REQ-SH4: The customs shapes can be linked to a body object and their behaviour will be controlled by the physics module through method Override.

REQ-BA1: A background is a set of sf::Shape that are linked together in a single custom object defined by a class

REQ-BA2: The background class constructor allows the user to pass the following parameters: color, shapes, main shape, physics manager, sound

REQ-BA3: The background class' main shape will be on the first layer and thus be at the bottom of the superposition stack

REQ-BA4: The background class' shapes are superposed in the order that they have been instantiated

REQ-BA5: The background class' variables cannot be altered after having been instantied. The user must create a new object.

5 Other Nonfunctional Requirements

5.1 Performance Requirements

Physics management must fast and able to support thousands of objects.

<If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.>

5.2 Safety Requirements

<Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product's design or use. Define any safety certifications that must be satisfied.>

5.3 Security Requirements

<Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

5.4 Software Quality Attributes

<Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use over ease of learning.>

5.5 Business Rules

<List any operating principles about the product, such as which individuals or roles can perform which functions under specific circumstances. These are not functional requirements in themselves, but they may imply certain functional requirements to enforce the rules.>

6 Other Requirements

<Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.>

6.1 Appendix A: Glossary

computer is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format. 12

graphics library is a library the provides a window and functions to draw to the window. Examples of graphics libraries are SFML and SDL2. 5, 12

<Define all the terms necessary to properly interpret the SRS, including acronyms and abbreviations. You may wish to build a separate glossary that spans multiple projects or the entire organization, and just include terms specific to a single project in each SRS.>

6.2 Appendix B: Analysis Models

<Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship a single project in each SRS.>

6.3 Appendix B: Analysis Models

<Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams.>

6.4 Appendix C: To Be Determined List

<Collect a numbered list of the TBD (to be determined) references that remain in the SRS so they can be tracked to closure.>