Software Requirements Specification

Berserk Engine Rendering System

Author: Orachyov Egor

Created: 10.03.2019 14:00
Last edited: 10.03.2019 21:54

Table of Contents

Introduction

This document represent software requirements specifications for Berserk Engine subsystem named Render System (or Rendering Engine).

Purpose

The purpose of this document is to give a detailed description of the requirements for the «Berserk Engine Rendering System» module. It will illustrate the purpose and complete declaration for the development of system. It will also explain system constraints, interface and interactions with other external applications. This document is primary intended to be exhaustive source of information for developer for programing first version of the system and for teacher (mentor), who will accept that work.

Scope

«Berserk Engine» is software game development kit (lib) designed for creating high performance 3D graphics applications with basic physics and audio effects. It should be used by programmers and software engineers, who want to develop their own game application.

This software kit is supposed to be free to download, share and use. Also, it is open source project with shared data base, which could be downloaded from GitHub.

«Rendering System» of the Berserk Engine is a Graphics Core and the most powerful and important part of the engine. It should provide features for rendering objects and applying effects (post process effects) in soft real time mode.

Definitions, acronyms, and abbreviations

Term	Definition
Engine / Game Engine	Software development kit - fully featured, developed as one or several .h include and .c/.cpp library files project, which provides functionality for creating graphics/physics/audio application (games, presentations, simulators, etc.).
Renderer / Rendering System	Game Engine subsystem, which is responsible for generating animated and rasterized in any way 3D/2D graphics.
User	Programmer, software engineer, in this context, user of the Engine framework and libraries.
Pipeline	Graphics generation and rasterization pipeline - the number of stages, steps, needed to generate single image with chosen quality, effects usage, time limitation and format.
Platform	Named device driver, with provides interface for access functions of hardware accelerated graphics processing unit.
OpenGL	Open Graphics Library (OpenGL) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.
XML	Format of creating text (human readable) files based on xml specification.
Texture	1, 2, 3 dimensional array of integer, floating point, short, signed, unsigned data, which could be interpreted as an image or other special purpose/format data. Could be created/loaded from image data, stored in common formats, such as png, bmp, jpeg, etc.
Material	Data structure, which stores properties of the abstract surface, which defines how rendered mesh will actually look in the rasterized/generated image.
Mesh	Raw vertex data: array vertices with indices, where each vertex stores data about some 2D/3D dimensional point. This data could be filled with: position, normal, tangent, bitangent, texture data of the point.
Model	An abstraction for representing 3D/2D dimensional objects in Rendering System. It stores mesh(s) and associated with them materials to perform correct representation of the model in rendered images.

Table 1 - Definitions

References

[1] «Game Engine Architecture», 2nd ed. by Jason Gregory

[2] «Learn OpenGL», by Joey de Vries

[3] «Real-Time Rendering», 4rd ed. by Tomas Akenine-Möller, Eric Haines and Naty Hoffman

[4] «The OpenGL Shading Language», 4th ed. by David Wolff

[5] «Real-Time Graphics Rendering Engine», 1st ed. By Hujun Bao, Wei Hua

[6] «ShaderX3: Advanced Rendering with DirectX and OpenGL», by Wolfgang Engel

Overview

The remainder of this document includes three chapters. The second one provides an overview of the system functionality and system interaction with other systems. Chapter also mentions the system constraints and assumptions about the product.

The third chapter provides the requirements specification in detailed terms and a description of the different system interfaces. Different specification techniques are used in order to specify the requirements more precisely for different audiences.

The fourth chapter deals with the prioritization of the requirements. It includes a motivation for the chosen prioritization methods and discusses why other alternatives were not chosen.

Overall description

This section will give an overview of the whole engine system. That structure will be explained in its context to show how the system interacts with other subsystems and introduce the basic functionality of it.

Product perspective

Rendering System specification will be provided accordingly to the overall Berserk Engine structure. The whole framework will consist of several huge subsystem parts, which are depicted in the diagram 1.

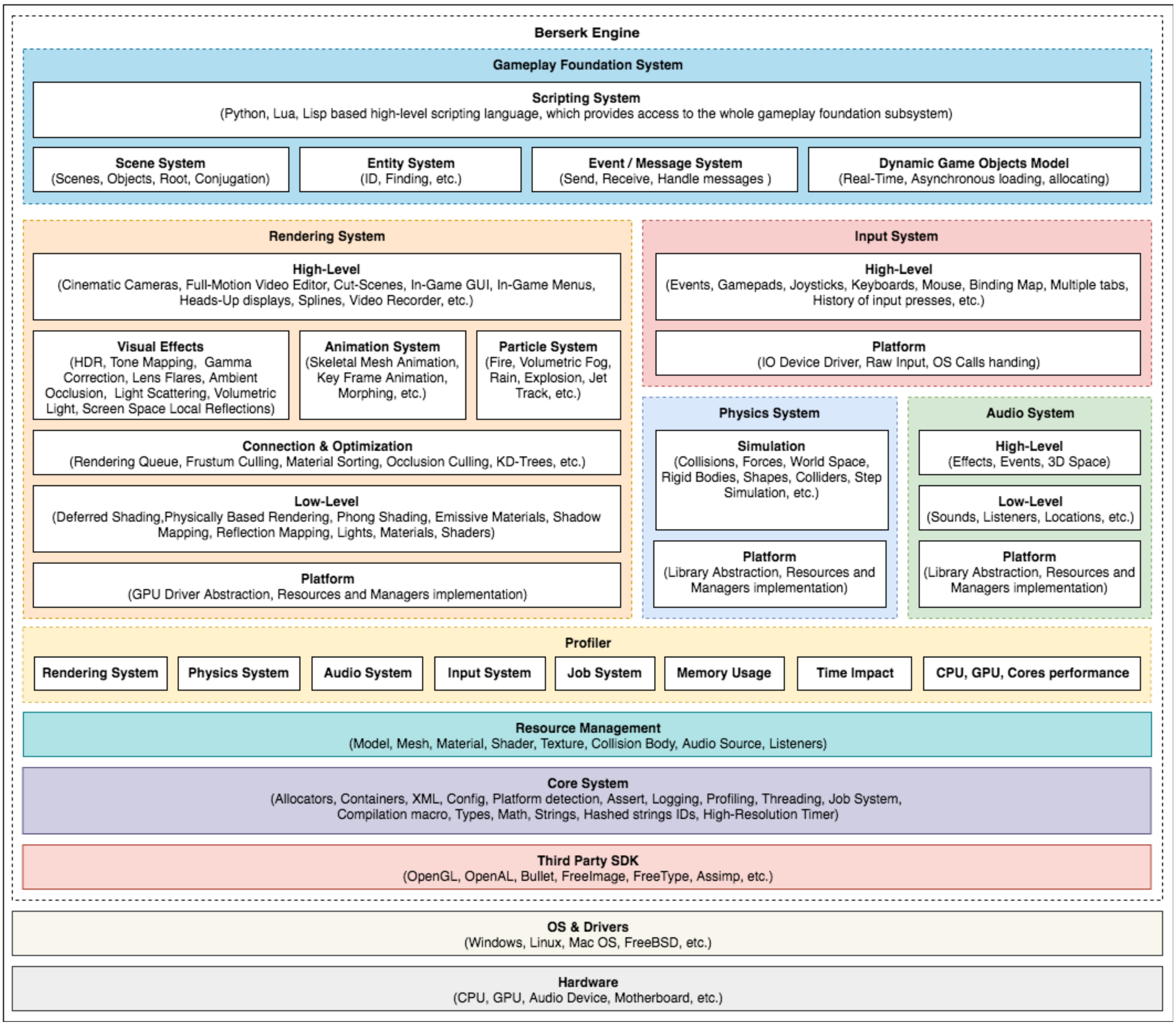
Berserk Engine
Gameplay Foundation System
Scripting System
(Python, Lua, Lisp based high-level scripting language, which provides access to the whole gameplay foundation subsystem)
Scene System
(Scenes, Objects, Root, Conjugation)
Entity System
(ID, Finding, etc.)
Event / Message System
(Send, Receive, Handle messages)
Dynamic Game Objects Model
(Real-Time, Asynchronous loading, allocating)
Rendering System
High-Level
(Cinematic Cameras, Full-Motion Video Editor, Cut-Scenes, In-Game GUI, In-Game Menus, Heads-Up displays, Splines, Video Recorder, etc.)
Visual Effects
(HDR, Tone Mapping, Gamma Correction, Lens Flares, Ambient Occlusion, Light Scattering, Volumetric Light, Screen Space Local Reflections)
Animation System
(Skeletal Mesh Animation, Key Frame Animation, Morphing, etc.)
Particle System
(Fire, Volumetric Fog, Rain, Explosion, Jet Track, etc.)
Connection & Optimization
(Rendering Queue, Frustum Culling, Material Sorting, Occlusion Culling, KD-Trees, etc.)
Low-Level
(Deferred Shading,Physically Based Rendering, Phong Shading, Emissive Materials, Shadow Mapping, Reflection Mapping, Lights, Materials, Shaders)
Platform
(GPU Driver Abstraction, Resources and Managers implementation)
Input System
High-Level
(Events, Gamepads, Joysticks, Keyboards, Mouse, Binding Map, Multiple tabs, History of input presses, etc.)
Platform
(IO Device Driver, Raw Input, OS Calls handing)
Physics System
Simulation
(Collisions, Forces, World Space, Rigid Bodies, Shapes, Colliders, Step Simulation, etc.)
Platform
(Library Abstraction, Resources and Managers implementation)
Audio System
High-Level
(Effects, Events, 3D Space)
Low-Level
(Sounds, Listeners, Locations, etc.)
Platform
(Library Abstraction, Resources and Managers implementation)
Profiler
Rendering System
Physics System
Audio System
Input System
Job System
Memory Usage
Time Impact
CPU, GPU, Cores performance
Resource Management
(Model, Mesh, Material, Shader, Texture, Collision Body, Audio Source, Listeners)
Core System
(Allocators, Containers, XML, Config, Platform detection, Assert, Logging, Profiling, Threading, Job System, Compilation macro, Types, Math, Strings, Hashed strings IDs, High-Resolution Timer)
Third Party SDK
(OpenGL, OpenAL, Bullet, FreeImage, FreeType, Assimp, etc.)
OS & Drivers
(Windows, Linux, Mac OS, FreeBSD, etc.)
Hardware
(CPU, GPU, Audio Device, Motherboard, etc.)

Diagram 1 - Engine Architecture

Core System will provide common functionality for all engine modules, systems components. It is designed as self-sufficient module built on top to the Operation System and Driver layer.

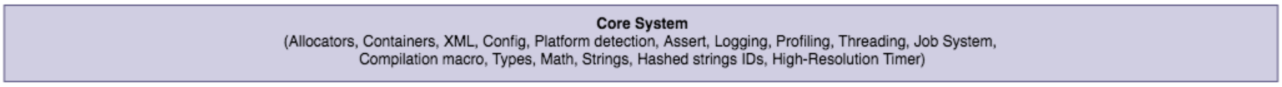
Core System
(Allocators, Containers, XML, Config, Platform detection, Assert, Logging, Profiling, Threading, Job System, Compilation macro, Types, Math, Strings, Hashed strings IDs, High-Resolution Timer)

Diagram 2 - Core System

Resource Management System evolves as framework part, which responsible for all type of operations and manipulations, made with engine resource. It involves: loading (with support of special and common formats of files), creating new ones, reference counting, proper memory allocation without fragmentation or out of memory exceptions. Also, in terms of SMP (Symmetric multiprocessing) and modern x86-64 platforms with Intel and AMD processors, it should provide sufficient and thread-safe access for engine resource in case of multithreaded execution.

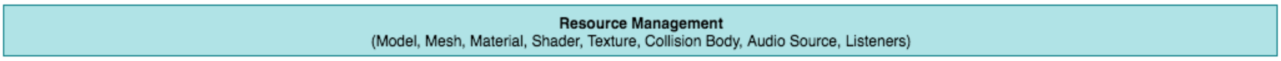
Resource Management
(Model, Mesh, Material, Shader, Texture, Collision Body, Audio Source, Listeners)

Diagram 3 - Resource Management

Profiling System designed as number of methods and data containers to collect, transfer, show and analyze execution time, load, memory usage, order of task competing and etc. in order to get information about bottlenecks or performance falls in systems/modules/stages in time of execution/simulation.

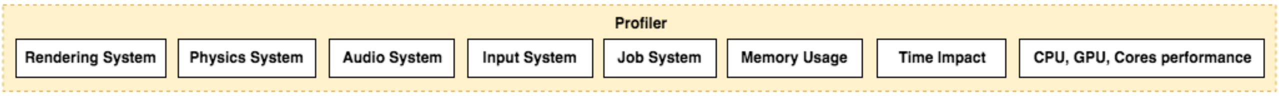

Diagram 4 - Profiler

Rendering System (or *Render System*) takes the most significant place in the whole engine architecture as the main visualization / rasterization module. It designed as number of layouts, where each module depends on the modules of lower level. Basic modules are Platform abstraction and Low-Level renderer, which define a way, in which engine interact with GPU of the target machine and common and usual rendering primitives, such as Light Sources, Materials, Shading models and etc. Higher level defines advanced rendering technics, Visual Effects, Animations and various Particle Systems with an ability to build customize a new one. High-Level Renderer is designed to be implemented on the top of the above mentioned modules as a high-level interface access to the whole rendering engine functionality with additional features, such as Cinematic Cameras, Cut-Scenes, GUIs, Video Editors and etc.

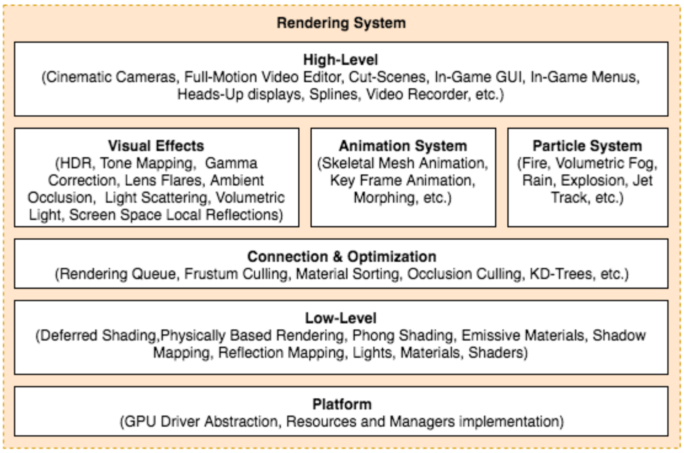
Rendering System
High-Level
(Cinematic Cameras, Full-Motion Video Editor, Cut-Scenes, In-Game GUI, In-Game Menus, Heads-Up displays, Splines, Video Recorder, etc.)
Visual Effects
(HDR, Tone Mapping, Gamma Correction, Lens Flares, Ambient Occlusion, Light Scattering, Volumetric Light, Screen Space Local Reflections)
Animation System
(Skeletal Mesh Animation, Key Frame Animation, Morphing, etc.)
Particle System
(Fire, Volumetric Fog, Rain, Explosion, Jet Track, etc.)
Connection & Optimization
(Rendering Queue, Frustum Culling, Material Sorting, Occlusion Culling, KD-Trees, etc.)
Low-Level
(Deferred Shading,Physically Based Rendering, Phong Shading, Emissive Materials, Shadow Mapping, Reflection Mapping, Lights, Materials, Shaders)
Platform
(GPU Driver Abstraction, Resources and Managers implementation)

Diagram 5 - Rendering System

Product functions

This paragraph will illustrate and declare the features and functionality of the Rendering System in Berserk Engine, as the part, which is considered to be implemented in this Requirement Specification.

Users (note: programmers) should be able to use rendering system (or system in this case) via its interfaces as an third party SDK, without having any idea: how this works. They should be able to send rendering requests to the system, specify desired quality of the generated images, configure effects, transform or/and manipulate models, load, modify, materials, be able to load different types of texture/mesh data from different formats and etc.

Also, system must be thread-safe in terms of multicore execution, except memory leaks and fragmentation, therefore, provide stable and self-sufficient mechanism of resource management with reference counting and loading/saving (in XML, for example).

Moreover, system should provide significant performance and work in real-time mode, because it is one of the main requirement to the 3D graphics games.

User characteristics

Users are considered to programmers or software engineers, who have significant knowledge in the field of graphics programming, have an understanding of underlying processes in rendering system, able to read documentation and code in C++ of version 11 or higher.

Constraints

Performance - rendering system should be able to rasterize images in 25 FPS mode, it is minimal required frequency of frames generation in 1280 x 720 (HD) resolution on Intel HD Graphics Video Card with 1.5 GiB Video RAM and 40 unified pipelines, with scene complexity of 10 totally applied materials and 100 models, rendered in Phong shadowing with 1000 vertices each.

Memory - all the loaded resources for 10 materials and 100 models and other resources (such as shaders, frame buffers, depth/reflection maps, etc.) should acquire less than 500 MiB of Video RAM.

Platform [future] - rendering system library should be able to be compiled at least on these specified platforms: Mac OS, Linux, Windows.

Assumptions and dependencies

Rendering system (first version of that) requires the following dependencies: OpenGL, GLEW, GLFW, FreeImage, FreeType, Assimp, MMX, SSE as third party libraries.

Apportioning of requirements

In the case that the project is delayed, there are some requirements that could be transferred to the next version of the application