Software Architecture Specification

BERSERK Engine

Author: Orachyov Egor
Created: 8.04.2019 20:00
Last edited: 10.04.2019 19:24
Last pdf gen: 10.04.2019 19:26

Table of Contents

Introduction

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

Purpose

The purpose of this document is to give a detailed description of the «Berserk Engine» (or engine) architecture. It will illustrate the system concepts, main process, system modules and their interaction. Also it will explain how the engine works with target platform OS, how uses its resources and memory (!). 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 (library) 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. As an model rasterization module, it must support modern and advanced rendering techniques (ambient occlusion, local reflections, light scattering, volumetric light, lens flares, , filmic tone mapping, …) as well as common shading models (raw, Phong, PBR).

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.
RAM	Random access memory available for CPU frequent read/write operations (store and use).
VRAM	Video random access memory available for GPU frequent read/write operations (store and use). Allows CPU to transfer data to/from VRAM from/to RAM for its usage by GPU.
GPU	Graphics processing unit, hardware graphics accelerator.
CPU	Central processing unit, main computing unit of target machine.
Cube Map	Special texturing technique, which allows to access texture data as 3D unit length box.
Reflection Map	Cube Map, which stores environment data, image, to create reflections on top of the model.
Depth Map	Texture, which has information, needed for creating real-time dynamic shadows and volumetric light effect.
Frame Buffer	Target buffer, driver option, which allows to generate image not only into window display, but in chosen frame buffer on GPU.
Uniform Buffer	GPU buffer which allows store/update/send data to shader. Reduces huge time impact on CPU / GPU memory transfer buffer.
GPU Buffer	Buffer, stored in GPU VRAM and used as indexed array for getting raw data about mesh/model vertices, texture coordinates and etc.
Shader	Program executed by highly parallelized GPU. In terms of rendering engine - it number of small (C style like) programs, which linked in one unit and executed as one program at any time moment of GPU work.
Phong Model	Default and common model (number of techniques) to create visual appearance of model from raw data.

PBR	Advanced physically based rendering technique, designed to be more realistic in terms of light physic world.

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 four chapters. The second one provides an overview of the system architecture - concepts, main design patterns, software and hardware components.

The third chapter provides the overview for engine system components, their interaction, data transfer, communication with third-party modules.

The fourth chapter deals with the memory management system. It illustrates main allocation patterns and strategy. Also shows how the engine world streaming systems allocates its data for continuous levels' regions loading.

The fifth chapter provides overview for rendering system, describes one rendering pass process, shows main concepts and used techniques.

Architecture overview

This section will provide an overview of the engine system architecture, describe some basic concepts and introduce common modules, needed for implementation.

Concept

Core engine architecture follows Entity-Component-System design pattern (Diagram 1), which means that the system has clear separation for data and systems, which will process this data. Accordingly to the ECS pattern, the main object is entity. It is an abstract item, which does not has world transformation, position, scale and default appearance. It only allows to attach components and process them in some way. Component is a data storage or some behavior, which shows/defines/declares how the entity will appear on the scene, how it will interact with other entities. More precisely, component could define transformation, physics properties, audio playing possibilities and visual appearance. In this model system is an object which knows how to process entities and components of chosen type, how to store and allocate it.

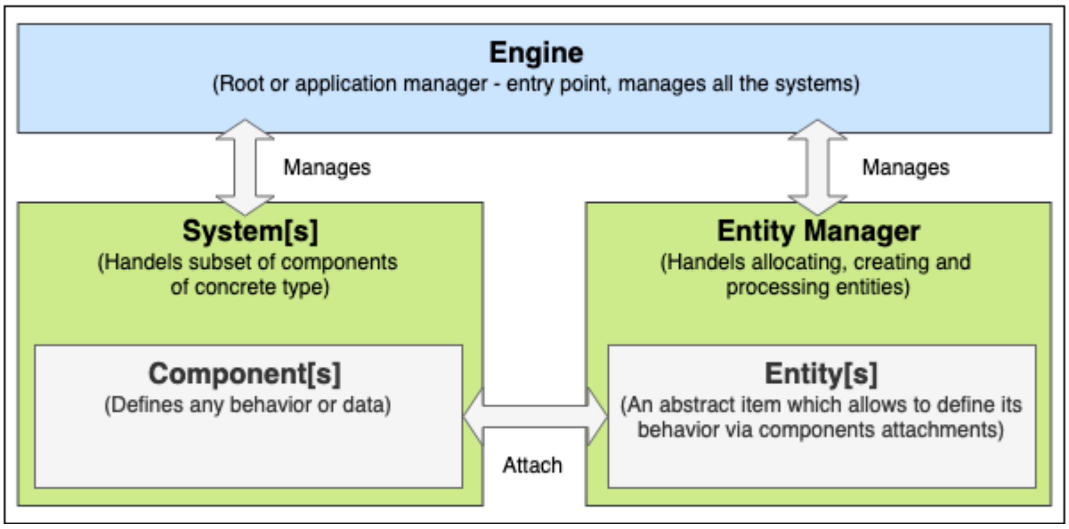
Engine
(Root or application manager - entry point, manages all the systems)
Manages
Manages
System[s]
(Handels subset of components of concrete type)
Entity Manager
(Handels allocating, creating and processing entities)
Component[s]
(Defines any behavior or data)
Entity[s]
(An abstract item which allows to define its behavior via components attachments)
Attach

Diagram 1 - Entity component system concept

Note: fully detailed class and components diagram will be provided later in the next sections of this document.

Model

The whole framework will consist of several huge subsystem parts, which are depicted in the diagram 1.

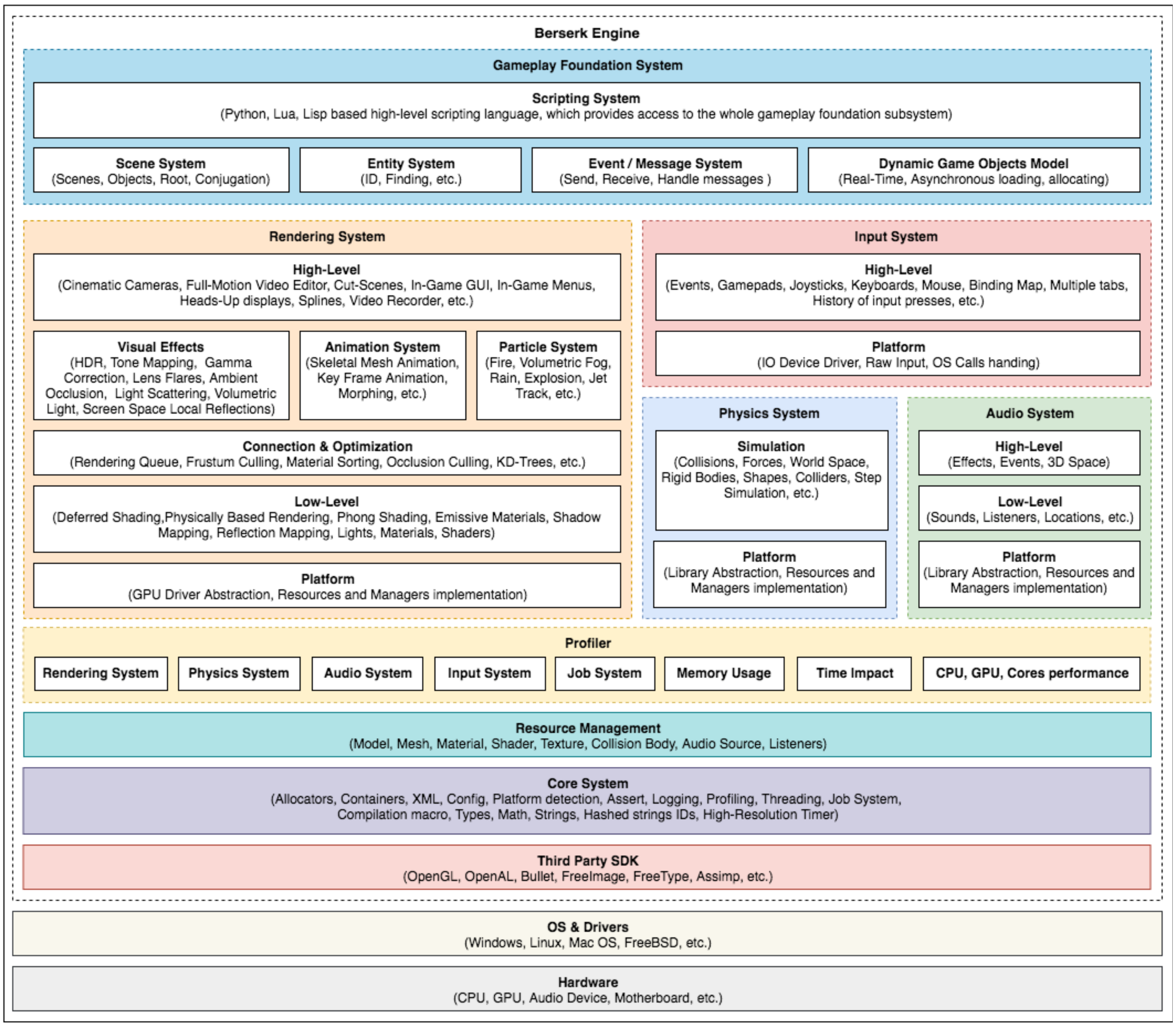
Berserk Engine
Gameplay Foundation System
Scripting System
(Python, Lua, Lisp based high-level scripting language, which provides access to the whole gameplay foundation subsystem)
Scene System
(Scenes, Objects, Root, Conjugation)
Entity System
(ID, Finding, etc.)
Event / Message System
(Send, Receive, 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 2 - Engine Architecture

Engine structure diagram is designed in module way/structure. Each module depends on modules of lower level and does not affect (or has minimal impact) on neighbor modules/parts. Also, because of the properties of this pdf file, Input system is placed on top of Physics System and Audio System, however, its is only placement requirement - in real system it does not depend on this mentioned modules.

Software components
Accordingly to the diagram 2 the engine has clear third-party components interaction. It involves OpenGL graphics specification driver of version 4.1 of higher, OpenGL Shading Language (GLSL) of version 4.1 or higher for building configurable rendering pipeline in Render System. Also it includes Assimp library for importing mesh/model data in runtime mode, Bullet physics as main physics component, Rapid XML parser for loading/saving xml files, which used for serialization of components/entities, FreeImage and FreeType importers for loading images (png, bmp) and fonts (ttf - primary) respectively.

Hardware components
For better understanding of underlying engine processes there is a need of target machine hardware specification. Common and the most popular format is depicted below (Diagram 3).

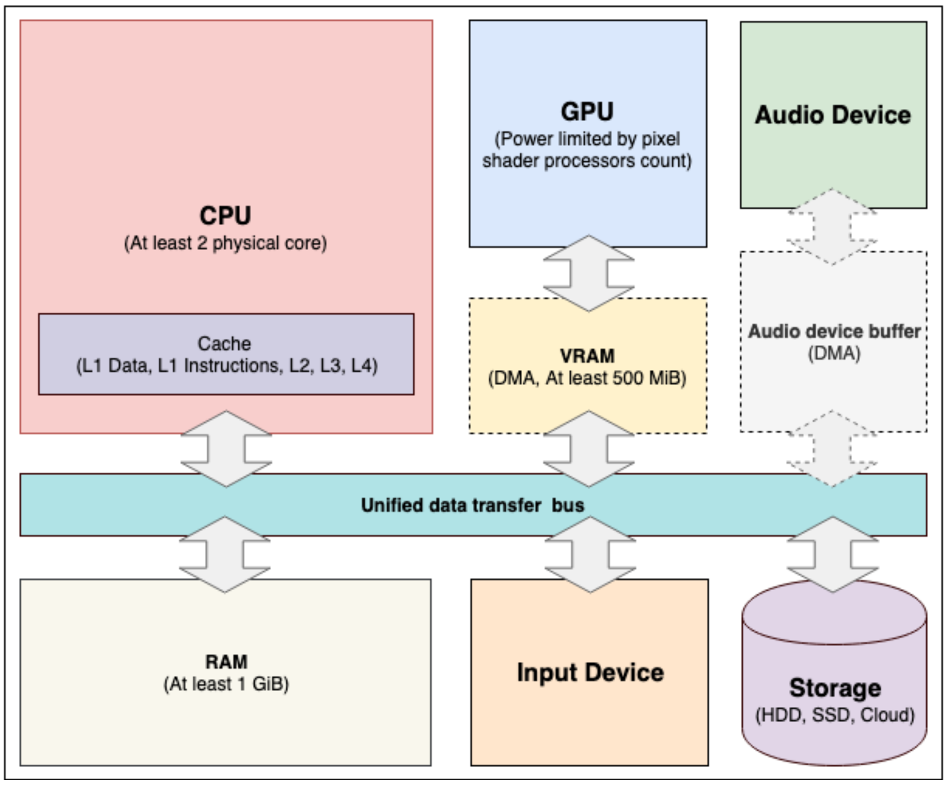
CPU
(At least 2 physical core)
Cache
(L1 Data, L1 Instructions, L2, L3, L4)
GPU
(Power limited by pixel shader processors count)
Audio Device
VRAM
(DMA, At least 500 MiB)
Audio device buffer
(DMA)
Unified data transfer bus
RAM
(At least 1 GiB)
Input Device
Storage
(HDD, SSD, Cloud)

Diagram 3 - Target platform hardware architecture

Above mentioned image is intended shows main principles of organizing common x86-64 based processor architecture personal computer (Except consoles, such as PlayStation 4 and Xbox One). In order to provide high performance in real-time mode engine architecture concepts should fit in current platforms design: cache of levels L1 - L4, slow transfer between RAM and VRAM, time-consuming data reading from HHD and SSD, multiple-cores execution with probable race-conditions and deadlocks.

System Components

Overview

Third-party dependancies

Memory Management

This section is intended to describe how the engine manages allocated memory, which concepts and techniques are used to avoid to expensive system calls malloc/free.

Strategy

The main idea of the allocation strategy is depicted in the following image (Diagram 4). We try to categories allocated memory in two types: memory, used by long-living systems and used by streamed levels' chunks.

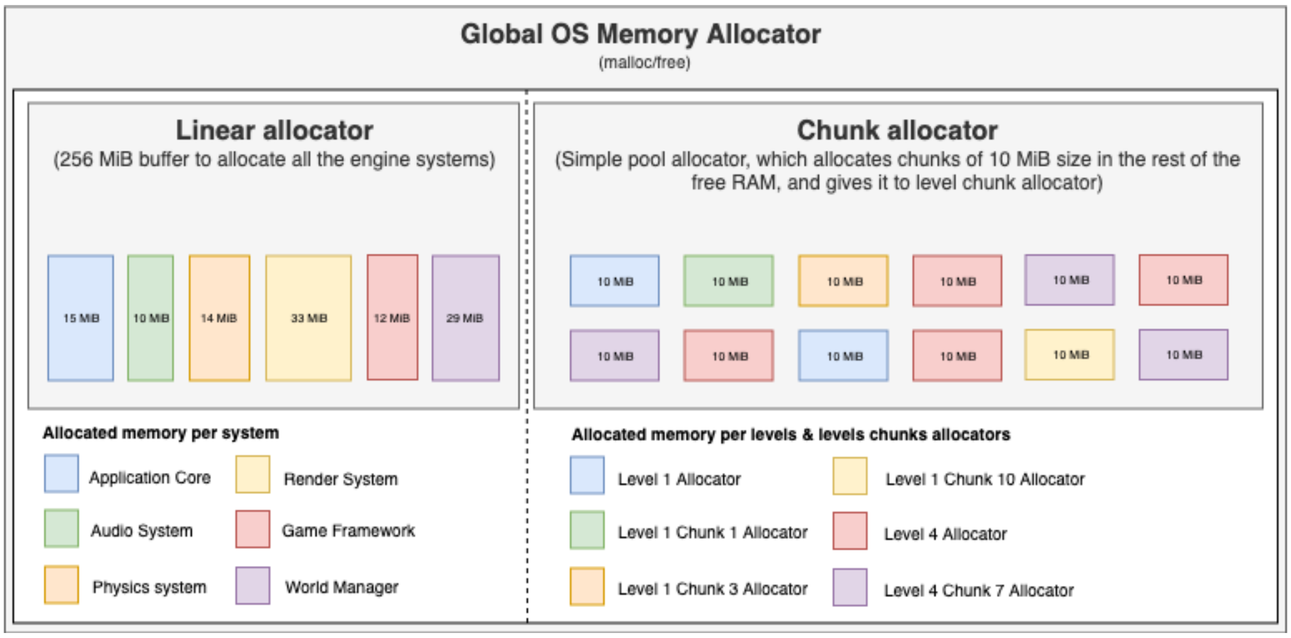
Global OS Memory Allocator
(malloc/free)
Linear allocator
(256 MiB buffer to allocate all the engine systems)
15 MiB
10 MiB
14 MiB
33 MiB
12 MiB
29 MiB
Chunk allocator
(Simple pool allocator, which allocates chunks of 10 MiB size in the rest of the free RAM, and gives it to level chunk allocator)
10 MiB
Allocated memory per system
Application Core
Render System
Audio System
Game Framework
Physics system
World Manager
Allocated memory per levels & levels chunks allocators
Level 1 Allocator
Level 1 Chunk 10 Allocator
Level 1 Chunk 1 Allocator
Level 4 Allocator
Level 1 Chunk 3 Allocator
Level 4 Chunk 7 Allocator

Diagram 4 - Memory management strategy

First of all, Linear allocator behaves as a static memory region, it means that allocated by this manager memory will be used from the beginning of the application execution till it is ended. Moreover, system can acquire its memory in non sequent mode, use it for internal allocator, only one requirement: it must fit in the buffer of 256 MiB size.

Chunk allocator (in other scope it is Pool Allocator) allows to create Chunk Allocator Clients. These clients behaves as simples linear allocators, which allocates memory from fixed size buffers of 10 MiB size given by Chunk Allocator. When client run out of memory, it asks for new on block, and this process is repeated. When client is deleted, all the allocated memory by this client freed AT ONCE. Therefor, when world manager unloads level (or level chunk) form the engine it will free all the memory at once (not per object). When the level or level chunk are loaded in the engine, chunk allocator will create a new client for this region of the level, and then all the loaded objects will allocate memory by this client.

In global scope engine can access only OS malloc/free, which will allocate memory for Linear and Chunk allocators.

World streaming

Rendering System

Concept

Overview

Rendering pass