Software Requirements Specification

BERSERK Engine Rendering System

Author: Orachyov Egor
Created: 10.03.2019 14:00
Last edited: 22.03.2019 11:48
Last pdf gen: 22.03.2019 11:49

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 (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.
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 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.

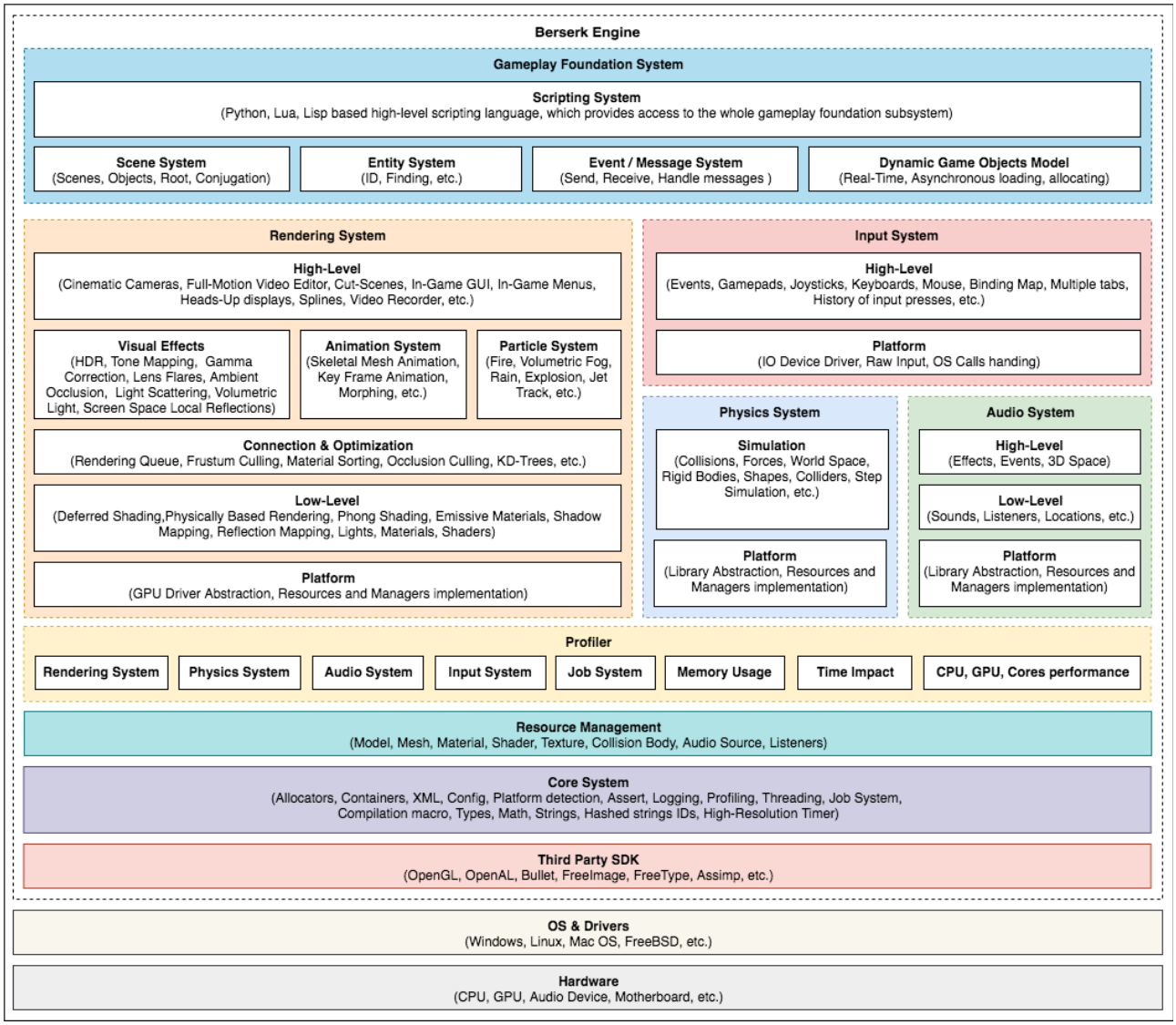


Diagram 1 - Engine Architecture

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.

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.

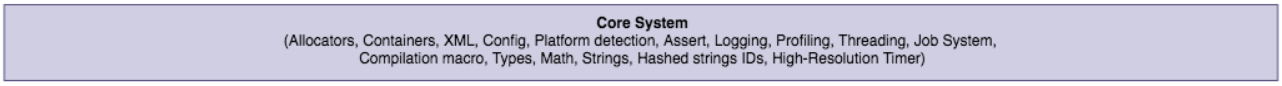


Diagram 2 - Core System

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.

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.

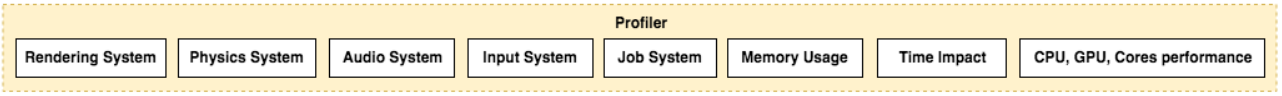


Diagram 4 - Profiler

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.

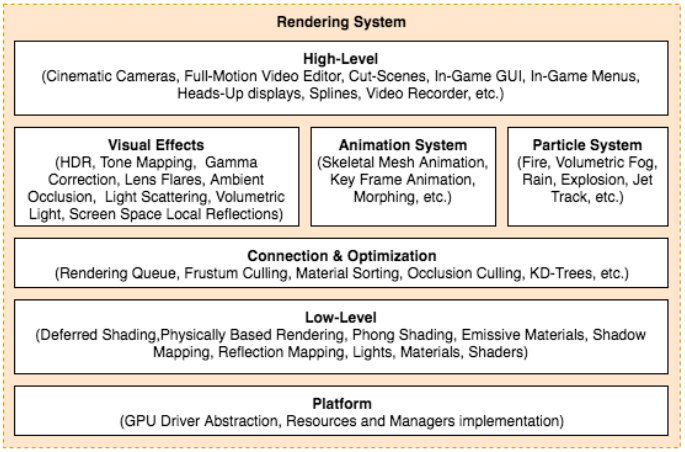


Diagram 5 - Rendering System

Product functions
This paragraph will illustrate and declare the features and functionality of the Rendering System in the 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 be Berserk Engine programmers/developers 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 and execution time will be limited by target platform hardware devices' power and possibilities. It impacts on scenes complexity as well as its detailing, shading model and number of applied post-process and pre-process effects. Also, memory limitations, such as amount of available free memory in RAM and VRAM, with bound of data-transfer time in BUS will have its own performance and quality constrains.

Rendering System is supposed to be designed and used only with C++ 11 programming language, it won't have GUI, build-in engine editor. Also, renderer won't provide another engine functionality, such as Audio Output, Physics, Scripting, Entity and Scene management, dynamic scenes loading.

First version of the engine must be implemented only with employment of OpenGL program layer, in minimal required case: without Animation system, Particle system and High-Level Renderer system.

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 or programmer has lack of time, there are some requirements that could be transferred to the next version of the rendering system.

Specific requirements

User interfaces

Note: *users* of Berserk Engine are game programmers, who want to create/manage 2D/ 3D graphics game, based upon this framework. *Users* of Rendering System are Berserk Engine developer(s), software engineers with significant and fundamental knowledge of engine architecture and 3D graphics theory.

Rendering system is a part of entire Engine system should provide sufficient interface methods/classes accordingly to the modules/components, mentioned in product perspective paragraph.

Each class/module must provide functionality in connection with concrete requirements, declared for each of the system. For example, hardware/driver abstraction layer Uniform Buffer class (see Code listing 1) could have the following format:

```
/**
 * Abstract uniform buffer object for cpu - gpu interconnection
 */
class GRAPHICS_API IUniformBuffer : public IResource
{
public:

    /** Create uniform buffer with chosen binding point, size and writes data in that */
    virtual void create(uint32 bindingPoint, uint32 size, const void *data) = 0;

    /** Update uniform buffer with data */
    virtual void update(uint32 size, const void *data) = 0;

    /** Update uniform buffer only after chosen offset */
    virtual void update(uint32 offset, uint32 size, const void *data) = 0;

    /** Bind uniform buffer to chosen binding point */
    virtual void bind(uint32 bindingPoint) = 0;

    /** Bind uniform buffer to its point */
    virtual void bind() = 0;

    /** Set fixed binding point for uniform buffer */
    virtual void setBindingPoint(uint32 bindingPoint) = 0;

    /** @return uniform buffer binding point */
    virtual uint32 getBindingPoint() = 0;

    /** @return Memory used at gpu side */
    virtual uint32 getGPUMemoryUsage() = 0;

};
```

Code listing 1 - Uniform Buffer interface

Resource interface could have the following format, showed in Code listing 2. First of all, interaction with resource components will be allowed via resource managers only, therefore its interface, as well as interface of platform graphics layer will be hidden and allowed to use only by engine developer(s).

An example of resource manager provided in Code listing 3 - it is Shader Manager interface for work with *shaders,* programs supposed to be executed on GPU side. One of remarkable features - it is loading from meta-info.xml files, which format will be listed below. It allows to programmer specify shader properties and loaded in the engine in one function call. Note: interaction with managers allowed only for engine programmers, not for final users.

```
/**
 * Interface for all CPU and GPU resources, which could be created
 * in the time of engine execution. This class, its inheritors
 * and associated resource managers is the part of the whole engine resource management system.
 * Note: that each resource requires explicit initialization and realese
 * methods call, so, only its manager can perform this. Therefore, direct usage
 * of resource by users is prohibited.
 */
class ENGINE_API IResource
{
public:

    /** Initializes empty resource instance with chosen name */
    virtual void initialize(const char* name) = 0;

    /** Add reference to resource if (f.e. you copy that) */
    virtual void addReference() = 0;

    /** Try to delete resource if its reference count == 0 */
    virtual void release() = 0;

    /** @return References count to this resource */
    virtual uint32 getReferenceCount() = 0;

    /** @return Memory cost of this resource (on CPU side only) */
    virtual uint32 getMemoryUsage() = 0;

    /** @return String name of resource (its unique identifier) */
    virtual const char* getName() = 0;

};
```

Code listing 2 - Resource interface

```
/**
 * An interface which provides access to the shader manager implemented in 3D
 * Rendering System. Responsible for creating and loading shaders in OS / Engine system.
 * Also provides reliable mechanism for importing shaders in real-time mode
 * from meta-info.xml files. Handles reference counting and memory allocation for resources.
 */
class ENGINE_API IShaderManager
{
public:

    /**
     * Explicit initialization of manager (must be invoked)
     */
    virtual void initialize() = 0;

    /** De-initialize manager */
    virtual void release() = 0;

    /** Rename chosen shader with new name */
    virtual void renameShader(IShader* shader, const char* name) = 0;

    /** Delete specified resource */
    virtual void deleteShader(IShader* shader) = 0;

    /** @return New empty (if it does not exist) shader with specified name */
    virtual IShader* createShader(const char* name) = 0;

    /** @return Shader with specified name */
    virtual IShader* findShader(const char* name) = 0;

    /** @return Pointer to resource with incrementing reference count */
    virtual IShader* getShader(const char* name) = 0;

    /** @return Shader from .mtl file with specified name (in path) */
    virtual IShader* loadShader(const char* path) = 0;

    /** @return Loaded shader with specified name from XML node */
    virtual IShader* loadShaderFromXML(const char* name, XMLNode& node) = 0;

    /** @return Memory usage on CPU (RAM) side */
    virtual uint32 getMemoryUsage() = 0;

};
```

Code listing 3 - Shader Resource Manager interface

Code listing 4 illustrates and example of XML configuration file for shader program importing in real-time mode into rendering engine system. It allows to specify paths to source code of program, define uniform variables, uniform blocks, specify subroutines and common implementations, also allows to vary program setup with reference to the underlying platform driver. It allows to get in execution time program, which behaves independently for each platform.

```
<!-- Specify one (or more) program(s) properties for loading     -->
<!-- Name attribute allows to find program in runtime via string -->

<?xml version="1.0" encoding="us-ascii"?>
<program name="Shadow map generation">

    <!-- Specify paths, uniforms and etc. for chosen -->
    <!-- graphics driver. Loader will manually check -->
    <!-- needed one and load appropriate shaders     -->

    <driver name="OpenGL">

        <!-- Specify shader type for code compilation and linking -->
        <!-- and relative or full path to the file on disk        -->

        <shader type="Geometry" path="{SHADERS}/ShadowMap/ShadowMap.geom" />
        <shader type="Vertex"   path="{SHADERS}/ShadowMap/ShadowMap.vert" />
        <shader type="Fragment" path="{SHADERS}/ShadowMap/ShadowMap.frag" />

        <!-- Specify uniform variables used in the shader -->
        <!-- Count - explicitly shows number of uniforms  -->

        <uniform name="SystemModel"   />
        <uniform name="LigthView"     />
        <uniform name="LigthPosition" />
        <uniform name="LightFarPlane" />

        <!-- Specify uniform blocks, evolved in shader program.   -->
        <!-- Name will be used to find block index, binding point -->
        <!-- specifies the uniform buffers and shader uniform     -->
        <!-- blocks binding properties (association)              -->

        <uniformblock name="UniformBlock1" binding="0" />
        <uniformblock name="UniformBlock2" binding="1" />
        <uniformblock name="UniformBlock3" binding="2" />

        <!-- Subroutines allows to vary functionality, used in  -->
        <!-- shaders via binding functions implementations to  -->
        <!-- specified function pointer, called subroutine     -->

        <subroutine name="LightningPass">
            <function> "Phong"     </function>
            <function> "PBR"       </function>
            <function> "Wireframe" </function>
        </subroutine>

    </driver>

</program>
```

Code listing 4 - Shader program meta-info.xml config

Rendering system will be integrated in the entire engine structure, therefore, final users, and engine programmers will be able to use engine features accordingly to the functional requirements specification, given bellow.

Hardware interfaces

Entire Berserk Engine framework with Rendering System is designed and supposed to be implemented for machines with processor architecture x86-64 with SSE extension and with integrated or discrete graphics accelerator built inside. Therefore, applications written with usage of its library should be able to be compiled and executed on specified platforms. Direct work with target CPU/GPU will be delegated to machine OS/Drivers, so framework will interact with needed resources via specified uniform API/ABI interface.

Software interfaces

Rendering System will interact and communicate with target machine GPU via C OpenGL interface (in first version). System will send to VRAM data from RAM, modify its data, read and write, as well as get it back from VRAM to RAM. Also, the following libraries support is required: FreeImage, FreeType, Assimp.

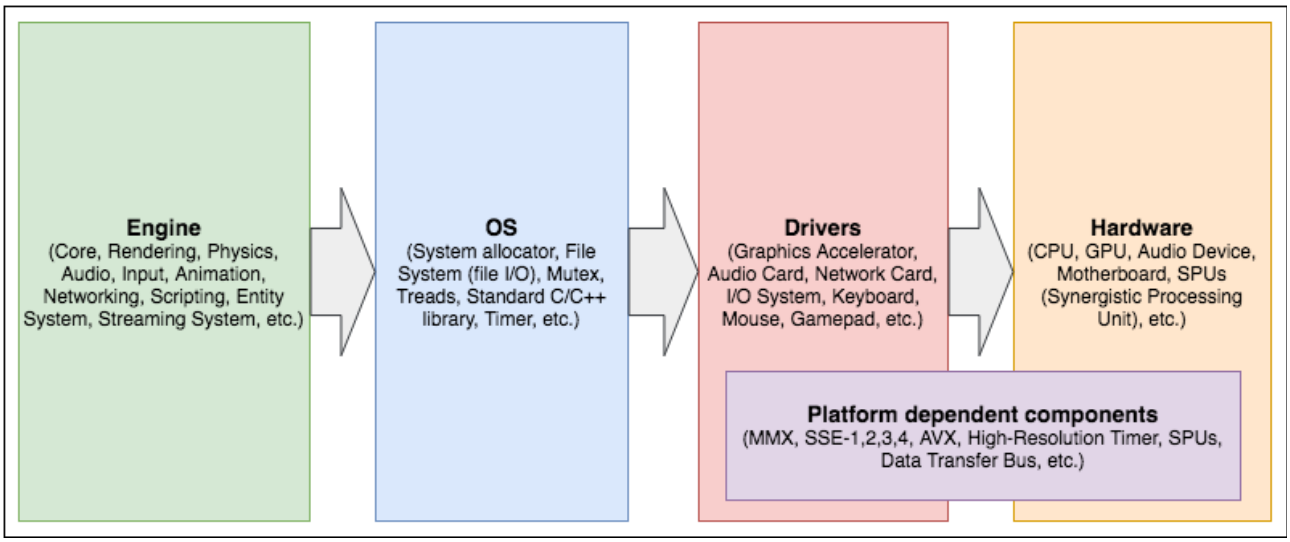


Diagram 6 - Interaction interface for Engine and Target machine

Functional requirements

Priority level: high

TITLE: Object
DESC: Default object, which could be created or serialized by the engine.

TITLE: Graphics accelerator abstraction layer
DESC: Virtual-class based interfaces for access CPU and VRAM execution resource with no dependance on target machine video driver (OpenGL, DirectX). Should be implemented as set of classes: IRenderDriver, IWindow, IFrameBuffer, IDepthBuffer, IGPUBuffer, IUniformBuffer, ITransformFeedbackBuffer, IInstantBuffer, IShader, ITexture, ISampler.

TITLE: OpenGL Support
DESC: Support for OpenGL of version 4.1 (minimal required) on MacOS for engine graphics abstraction layer.

TITLE: Component system
DESC: As part of all engine structure there is need of Component system: universal component object, which has own behavior and could be attached to the entity. Has the following properties: factory builder, type, name, on-create, on-update, on-delete events.

TITLE: Render Node
DESC: Resource-based render node, which encapsulates common and important rendering information/properties, needed for visualization and rasterization of models, vegetation, light sources, particle systems, water, etc.

TITLE: Indexed Mesh
DESC: Mesh with associated material, used as indexed object in model component. Should have own AABB for culling, parent, transformation, ability to be loaded from HHD.

TITLE: Model Component
DESC: Component-based model object, represented as array of indexed meshes with associated materials, which could be loaded as model from HHD FBX file and sent to the rendering queue as 1 entry.

TITLE: Light Component
DESC: Component-based light node, which represents 2D/3D space light source, which could be placed somewhere in 3D scene with chosen shading, rendering, lightning, emission properties.

TITLE: Camera Component
DESC: Component-based camera with view and occlusion properties of the scene. Provides information about all the camera effects: flares, depth of field, motion blur, view frustum, view distance, etc.

TITLE: Material
DESC: Resource based material object, which contains information about surface appearance and applied rendering technique for rasterization.

TITLE: Window (Display)
DESC: Support only for one active display for rendering image data.

TITLE: Renderin Engine Base
DESC: Base class with pointers (references) to all the rendering engine managers, and other static shared data: image importer, model importer, material manager, shader manager, buffer manager, rendering driver, etc.

TITLE: Resource management system
DESC: Number of resource managers for mentioned above resource types with the following features: loading materials, shaders, textures from specified xml files, saving textures, reference counting, creating, finding resources. Managers: Material manager, Buffer manager, Shader manager, Objects manager.

```
/**
 * An interface which provides access to the shader manager implemented in 3D Rendering System.
 * Responsible for creating and loading shaders in OS / Engine system. Also provides reliable
 * mechanism for importing shaders in real-time mode from meta-info.xml files.
 * Handles reference counting and memory allocation for resources.
 */
class ENGINE_API IShaderManager
{
public:

    /**
     * Explicit initialization of manager (must be invoked)
     */
    virtual void initialize() = 0;

    /** De-initialize manager */
    virtual void release() = 0;

    /** Rename chosen shader with new name */
    virtual void renameShader(IShader* shader, const char* name) = 0;

    /** Delete specified resource */
    virtual void deleteShader(IShader* shader) = 0;

    /** @return New empty (if it does not exist) shader with specified name */
    virtual IShader* createShader(const char* name) = 0;

    /** @return Shader with specified name */
    virtual IShader* findShader(const char* name) = 0;

    /** @return Pointer to resource with incrementing reference count */
    virtual IShader* getShader(const char* name) = 0;

    /** @return Shader from .mtl file with specified name (in path) */
    virtual IShader* loadShader(const char* path) = 0;

    /** @return Loaded shader with specified name from XML node */
    virtual IShader* loadShaderFromXML(const char* name, XMLNode& node) = 0;

    /** @return Memory usage on CPU (RAM) side */
    virtual uint32 getMemoryUsage() = 0;

};
```

Code listing 5 - Shader manager interface

TITLE: Engine sizer
DESC: Special sizer class with ability to accumulate the information about memory usage for all types of rendering engine manager (count RAM and VRAM memory acquirement). Interaction should be implemented via special function:

```
/* Profile memory usage on CPU and GPU size */
virtual void getMemoryUsage(MemorySizer& mSizer) = 0;
```

Code listing 6 - Memory profiling

Priority level: medium

TITLE: Entity
DESC: Basic scene tree node - entity, object, with AABB, components list and list of child entities (behavior assumption). Each entity should have factory methods with serialization features and xml loading support.

TITLE: Rendering Queue
DESC: Queue with ability to submit rendering node for rendering in different modes.

Mode name	Description
Background	Objects rendered before any geometry on the scene. After that stage the depth buffer will be cleared.
Geometry	Main geometry pass - render nodes sorted in the order of used materials. Also, allows to render models with customs shaders (later support).
Alpha blend	Geometry with materials that rendering involves alpha blending with previous color data in the currently rendered buffer. All the render nodes will be sorted in the order of camera distance (from far to near).
Overlay	Everything what should be rendered on top the result image with cleared depth buffer. It could be camera effects or UI elements.

Table 2 - Queue rendering modes (lines)

TITLE: Frustum Culling
DESC: Optimization technique to reject some render nodes in order to increase performance. Minimal requirement: do this in current thread with SSE, MMX intrinsics usage.

TITLE: Pipeline
DESC: Base rendering pipeline with utilization of mentioned rendering queue modes.

TITLE: Shadow mapping
DESC: Generating directional shadow maps for light sources and its mapping.

TITLE: Reflection mapping
DESC: Generating local reflection maps and its applying for reflective surface.

TITLE: Geometry Buffer
DESC: Generating Geometry buffer with all the information about the scene geometry for deferred rendering.

TITLE: Raw Shading Model
DESC: Support for shading objects with raw albedo color applying

TITLE: Blinn-Phong Shading Model
DESC: Support for Blinn-Phong shading

TITLE: Physically Based Shading Model
DESC: Support for PBR shading

TITLE: SSAO
DESC: Screen space ambient occlusion mag generation for screen space geometry with applying on final image

TITLE: Deferred shading
DESC: Shading for scene objects with different shading models, involves applying SSAO ambient value, sample shadow and reflection maps.

TITLE: Bloom Effect
DESC: Blooming bright areas of finally rasterized image.

TITLE: Tone Mapping
DESC: Default (later: adaptive cinematic) tone mapping (exposure, exp, Reinhard).

TITLE: HDR
DESC: Support for RGBA float-16 frame buffer with high range color values for display image.

Priority level: low

TITLE: SSLR
DESC: Screen space local reflections, applied to the totally rendered scene with

TITLE: Lens flares
DESC: Lens optics flares effect on main virtual rendering camera, visualized as textured glare on screen.

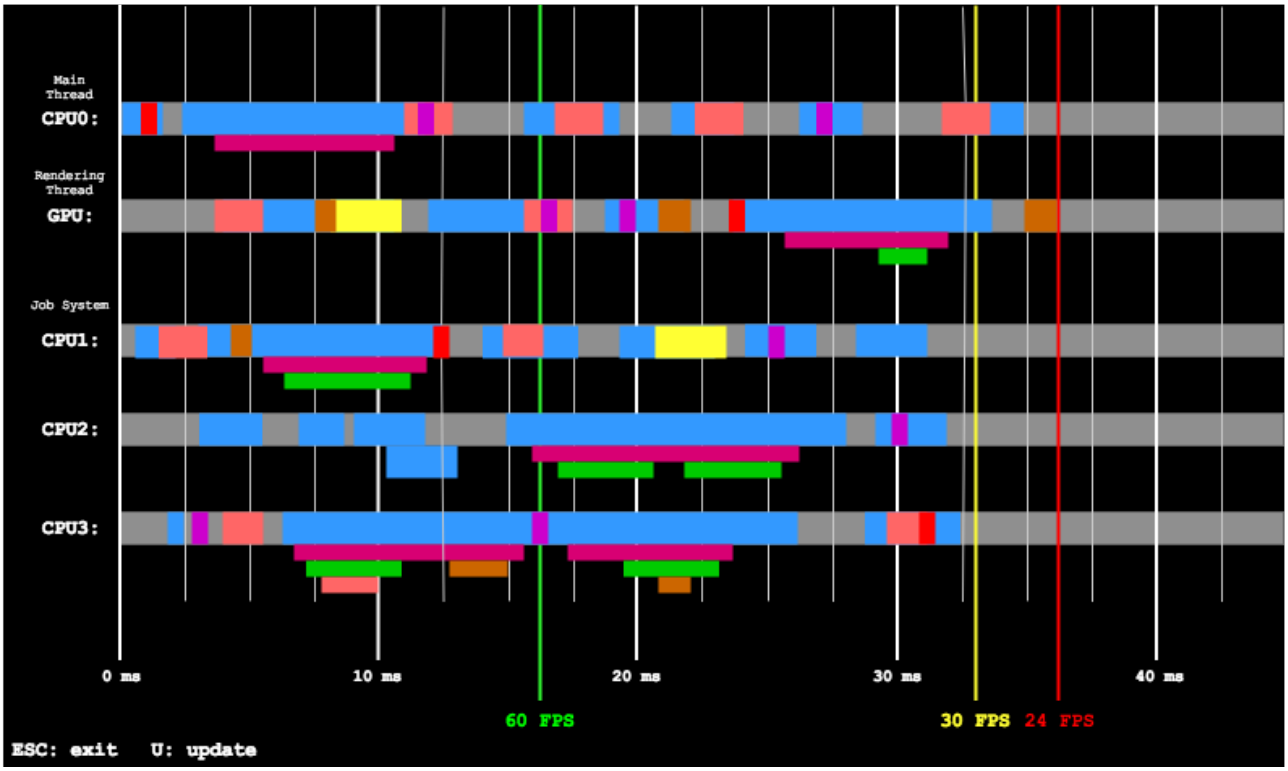
TITLE: Particle system
DESC: Base components for creating different systems, with default and custom cullers, emitters, bases, state machines, lifetimes. Main requirement: because of GPU performance limitations, particle systems should be created and handled on CPU side.

TITLE: Default Particle Systems
DESC: Default set of effects, such as fog, fire, explosion, ion engine trace, etc.

TITLE: Particle factory
DESC: Ability to compose own particle system from components, provided by factory.

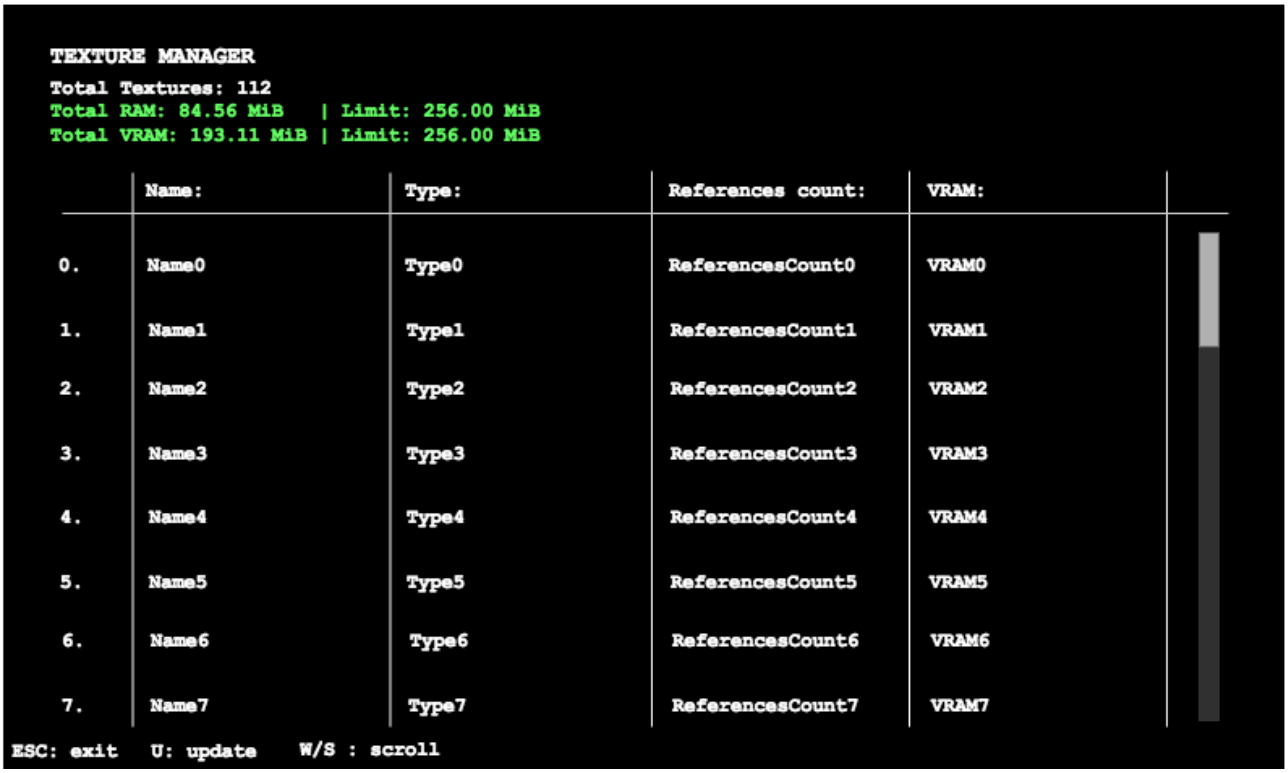
TITLE: Build-In On-Screen profiling tool
DESC: Subset of integrated into engine tools for profiling real-time application execution. It should provide information in on-screen mode, therefore, user does not have to exit the application to see the params of current execution. Tools should provide info about: threads load in terms of completed tasks and used time, memory usage, systems' memory usage, memory resource tables, tables of execution time for systems, etc.

The following picture illustrates a minimal required UI for threads execution profiler: it must provide information about acquired ms to complete tasks, mark tasks and threads idling, show current and desired fps rate, etc. Also tool could provide extra text description as optional help for user.



Picture 1 - Threads execution profiling tool

Picture 2 show an example of how could be structured texture manager memory usage table. In minimal configuration in could involve a list of the all loaded textures with its names, memory acquirement and reference count. Also it optionally could demonstrate total RAM and VRAM usage with color indication of specified limitations bounding. (In the example green color means that current memory usage less the specified limitations.)



Picture 2 - Texture manager memory acquiring profiling tool

Performance requirements

TITLE: Execution time
DESC: Number of frames per second in real-time rasterization mode in test scene
METER: FPS
MUST: More than 25 FPS
PLAN: Stable 30 FPS
WISH: More than 60 FPS
FPS: DEFINED: Frames per second

TITLE: RAM Memory usage
DESC: RAM Memory used to store all the allocated resources for rendering in test scene
METER: MB
MUST: Less than 1000 MB
PLAN: Less than 500 MB
WISH: Less than 250 MB
MB: DEFINED: Megabytes

Note: Target platform test machine properties:
CPU: Intel Core i7 4770 HQ 2,2 GHz
GPU: Intel Iris Pro Graphics
Display: 15,4 inch, Resolution 2880x1800 (Retina)
RAM: 16 GB 1600 MHz DDR3
VRAM: 1536 MB

Design constraints

TITLE: Display Resolution
DESC: Window display resolution of rendering for test scene
METER: Screen resolution in pixels
MUST: 640x480 (VGA)
PLAN: 1280x720 (HD)
WISH: 1920x1080 (Full HD)

TITLE: Hard disk driver memory usage
DESC: Memory used to store assets data for test scene
METER: GB
MUST: Less than 1.5 GB
PLAN: Less than 1 GB
WISH: Less than 0.5 GB
GB: DEFINED: Gigabytes

Note: Target platform test machine properties:
CPU: Intel Core i7 4770 HQ 2,2 GHz
GPU: Intel Iris Pro Graphics
Display: 15,4 inch, Resolution 2880x1800 (Retina)
RAM: 16 GB 1600 MHz DDR3
VRAM: 1536 MB

Software system attributes

TITLE: Reliability
DESC: If there is an undefined situation/behavior because of incorrect data/input code, system must provide sufficient information about error and shutdown the application
RAT: In order to get more information about crash/shutdown causes.

TITLE: Maintainability
DESC: System classes and components should be documented and sorted
RAT: In order to provide sufficient information about system and show how to use it

TITLE: Portability
DESC: System should be able to be built and executed on Linux, Mac OS platforms
RAT: In order to build cross-platform applications

Prioritization

Explicit prioritization of rendering engine modules' development is needed because of two main reasons: lack of development time and human resource (in case of student's project).

Also rendering engine software requirements were separated in order to get self-sufficient and working system as soon as possible in terms of development time.

Each module (block) depends on modules, developed earlier and placed lower in main structure diagram, therefore swopping of development order is impossible (now).

Finishing of each requirements module will be followed with new release to mark significant and important parts of rendering (and whole) engine development.