

**FIT DEPARTMENT**

Computer Science and System Design

Software and Hardware Development



**Software and Hardware Development**

**System Architecture Document for project #6: “Rendering”**

Date: 9.03.2020

Authors: Yaroslav Odintsov, Artem Plusnin, Ekaterina Dubinskaya.

1. Goals and limitations

1.1. Key functional requirements

Use-case ”Scene Import” – to import scene from the file.

Use-case ”Scene Save” – to save scene or image to the disk.

Use-case ”Render” – to start render algorithms.

Use-case “Edit” – to edit the scene.

1.2. Non-functional requirements

Environment:

Render should run smoothly on machines with operation system Windows.

For better performance recommends using Nvidia graphic adapter.

Reliability:

Render should work correctly with any user’s actions.

Render should fork for guarantee low time. [maybe tests]

Extensibility:

In our application only interface specifications are provided. That’s mean render can’t be extended.

1.3. Architectural goals

- The application must operate on Windows.

- The application must work with most of nVidia graphic adapter.

- The application must be easily extensible to support new 3D-types.

1.4. Additional goals, restrictions and preferences

- Using C++ to write render, due to its speed and compatibility with cuda.

- Using x3d format to study xml format, it can be very useful.

- Make a simple to understand UI using QT.

- Editor should be also fresh and simple to work fast with minor changes on the scene.

2. Goals analysis

2.1. Render goals

-It should work for guarantee low time and good quality of output.

Render uses the raytracing algorithm. Raytracing will make the output feel much more realistic than any other algorithm. It has a lot of optimizations for better performance. It also works much faster if user has a Nvidia graphic adapter because render has a CUDA part that works on GPU and it is much faster than rendering on CPU.

Light: Algorithm simulates the nature properties: a light source emits a ray of light which travels, eventually, to a surface that interrupts its progress.

Ray: Ray will be a straight line (ignoring relativistic effects). Any combination of four things might happen with this light ray: absorption, reflection, refraction and fluorescence.

Absorption: A surface may absorb part of the light ray, resulting in a loss of intensity of the reflected and/or refracted light.

Reflection: It might also reflect all or part of the light ray, in one or more directions.

Refraction: If the surface has any transparent or translucent properties, it refracts a portion of the light beam into itself in a different direction while absorbing some (or all) of the spectrum (and possibly altering the color).

Algorithm: When a ray hits a surface, it can generate up to three new types of rays: reflection, refraction, and shadow. A reflection ray is traced in the mirror-reflection direction. The closest object it intersects is what will be seen in the reflection. Refraction rays traveling through transparent material work similarly, with the addition that a refractive ray could be entering or exiting a material. A shadow ray is traced toward each light. If any opaque object is found between the surface and the light, the surface is in shadow and the light does not illuminate it. This recursive ray tracing added more realism to ray traced images.

2.2. Formats goals

- Application should support x3d format.

Parser can work with x3d format to import 3d scenes form other 3D-editing programs.

2.3. UI goals

- UI should be simple to understand

UI itself has typical design (like in every program that can fork with text files for example). It has only must-have buttons on regular for them places. There is no massive or extra stuff for giving the user no reason to confuse. UI is written using QT as a good tool for building small and simple designs.

2.4. Editor goals

- Editor should be easy to use and support minor editions on scene.

Editor is made as part of UI that makes user fast to understand it. In fact, Editor works with render data. We use KD-tree for editing scene and fast re-rendering it. As a part of UI it has Qt-written parts.

/\*[For each goal there must be overview of ways to solve it and analysis of their applicability, pros and cons

regarding the system to develop. The overview could mention external libraries, frameworks, algorithms,

known architectural solutions (architectural patterns), suggest some modifications of them. For each

external element provide reference to corresponded publications/documentation.

The rest of this section including subsections are project-specific and provided just as shortened examples.]\*/

2.1. [ex.] Security

Due to multi-user nature of the system, there must be session support, each session must be authenticated

and each operation must be authorized.

There are 3 main user roles in the system:

- Regular user, who can do the following: UC1, UC3, UC42

- Auditor, who can...

- Administrator...

Some users can have “regular user” and “auditor” roles simultaneously

There are two main approaches to support authorization:

- Protection rings [ref], that...

- Access control list [ref], that...

Because “auditor” and “regular user” roles have different sets of use-cases, protection rings approach is not

applicable, and ACL is the only option (It is not true in fact. There are some other approaches less complex

than ACL, so our architect missed important thing here).

In ACL the following operations must be covered: ...

Authentication can be implemented either via login/password pair stored in local DB and appropriate login

form, or via external service like Google or Facebook.

It is easier to delegate it to ready-to-use external service, so let’s use Google for authentication. (In fact, the

solution is questionable at least. First, there are usually other user-related data besides login/password that

must be stored on the application level anyway. Second, integration with an external general-purposes

service sometimes could be more complex than writing your own project-specific solution).

2.2. [ex.]Web-interface

...

2.3. [ex.]OS support

...

...

3. Solution description

A picture containing clock, object

Description automatically generated

[Provide overall description of the system’s architecture].

3.1. Modules and subsystems

[Identify top-level modules/subsystems. Pictures are welcome. For each module provide clear description of

its functions/responsibilities and interactions with other modules (contact in other words). Also, for each

module identify architectural decisions like key frameworks, libraries, languages, architectural patterns,

algorithms. References to the section 4 are OK]

3.2. Deployment

[Provide deployment view, i.e. which processes/code parts running on separate devices and how they

interact with each other. For distributed software by “devices” separate computers are assumed first of all.

Also, this section could show functional distribution between CPU and GPU, interactions with external

sensors or other specialized hardware. Pictures are welcome.]

3.3. ...

…

4. Key architectural elements

[Identify and define key system-wide (i.e. affecting multiple modules from section 3) behavioral elements like

protocols, algorithms, computational mechanisms. Usually such elements are corresponded to at least one

goal from the section 1. Each element must be described in terms of modules identified in section 3 and their

behavior. Pictures are welcome.

This section could be placed before section 3 if necessary.

The rest part of this section including subsections are examples]

4.1. [ex.]Authentication protocol

...

4.2. [ex.]Authorization

...

4.3. [ex.]Fault-tolerant network interaction

...

4.4. [ex.]Modules API

All the modules’ external APIs should follow Command-Query Separation [<reference>] pattern excluding ...,

where atomic types should be used and, thus, they must violate this pattern.

All the modules excluding ... must implement their APIs in two forms: 1) regular Java API in terms of

interfaces, and 2) RESTful API using Jetty [<reference>]as embedded servlet container.

Regular Java API must be used in most situations due to better performance and reliability. RESTful APIs

should be used mostly for automatic testing purposes.

4.5. [ex.]External plug-in API

For module ... the external API for 3rd party video processing plugins must be implemented in reliable way. A

plug-in must be run as a separate process restricted by RAM and CPU usage by ... So, the main application

shall tolerate the situations where plug-in killed due to critical error (segmentation fault or so), overusing

memory or CPU, spawning threads.

Interaction with plug-in on lower level must be implemented via STDIN/STDOUT/STDERR. On higher level,

the sandbox for plug-ins must be implemented (see 3.NN) and provide the following operations: ...

To transmit objects through the raw stream JSON serialization/deserialization must be used. Jackson

[<reference>] library is suggested for this. There is the requirement to possess bandwidth of ... MB/s for

plugins, so Jackson must be checked first to match this.

4.6. ...

…

5. Platform

Users’ system requirements (minimal):

- Personal computer with Windows.

- Processor: any.

- Graphic adapter: any.

- X Mb of free space on a disc.

Users’ system requirements (recommended):

- Personal computer with Windows 10.

- Processor: Intel Pentium G4560/Amd Ryzen 3 1200.

- Graphic adapter: (NOT AMD!) nVidia GTX1050Ti.

- X Mb of free space on a disc.

[Enumerate all the hardware and external software decisions that come from all the previous section.

All the rest is just an example.]

Server platform: x86-64/amd64, Linux 2.6

Client platform: ARM, Android; Web browser (Edge, Chrome, Firefox)

Language: Java on server and Android client, JavaScript on Web-based client

Server frameworks and libraries: Tomcat as servlet container, Spring (Boot, Security, MVC), PostgreSQL as

DBMS, Hibernate for interactions with DB, Jackson for JSON(de-)serialization, JMeter for load testing.

Web client frameworks and libraries: ReactJS