RENDERING OPTIMIZATION APPROACH FOR GAME ENGINE DEVELOPMENT

Olena Chebanyuk, Mykhailo Mushynskyi

Abstract: *The article is devoted to the problem of creating a software tool for optimizing rendering in game engines, which will help users to quickly and easily configure their projects. The tool provides a centralized simple interface to the rendering tools built into a game engine. The key possibility of the tool is the management of visual effects, such as Bloom, Volumetric Shadow, Ambient Occlusion, etc. There is also an opportunity to use profilers and get a critical information for project optimization.*

Keywords: *Rendering Optimization, Game Engine, Unreal Engine, GPU, Level of Detail, Ambient Occlusion, Depth of Field, Bloom, GPU Profiler, FPS, Quad Overdraw, Dynamic Light, Volumetric Shadow, Static Light, Post-Processing, Static Mesh.*

ITHEA Keywords: *B.8.2 Performance Analysis and Design Aids, D.2.3 Coding Tools and Techniques, D.4.8 Performance.*

DOI: *https://doi.org/10.54521/ijita28-02-p04*

Introduction

Every modern game engine contains numerous tools for customizing graphic effects and determining their impact on the performance of the gaming application. They are usually difficult to use and require a large amount of time to master. Sometimes, developers neglect such tools because of the long process of identifying and solving optimization problems.

Gaming forums often have complaints from gamers about poor game optimization. Sometimes, this can be due to the inexperience of developers,

especially in the case of young indie studios, although it also happens in AAA projects of large gaming corporations. This may also be due to a lack of funding or insufficient time to complete the product that is usually fixed later with patches and hotfixes.

Terminology of the proposed approach

The post-processing effects have the greatest impact on the rendering process. These include volumetric fog, dynamic lighting, texture filtering, shadow, particle systems, and bloom. Disabling any of the above will increase the processing speed of a game project by 2-10%. However, it is not necessary to turn them off as it is possible to adjust their complexity.

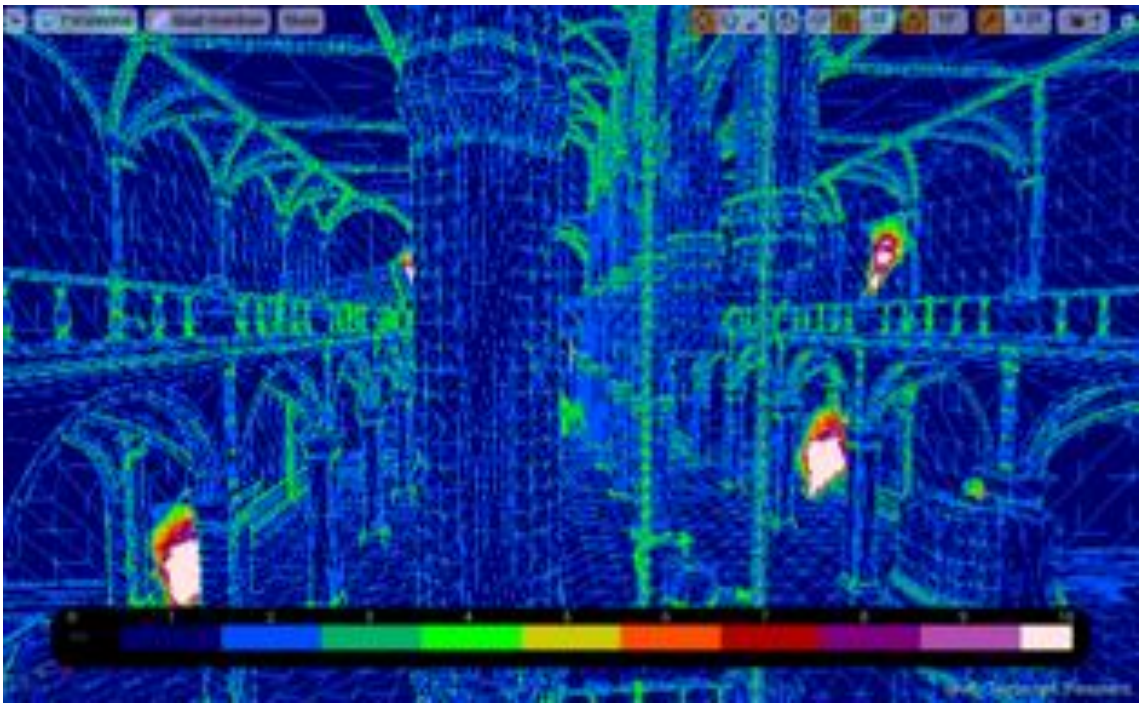

Figure 1. Unreal Engine quad overdraw detection tool.
Dark blue for low density, white – for extremely high

Objects complexity also has a strong effect on the processing time of a virtual scene. The term ***quad*** overdraw means a high concentration of overlapping pixels over each other. In other words, quad overdraw [Intel, 2017] occurs when

a large number of polygons are called to compute the same pixel. A popular way to solve this problem is to reduce the detail of static meshes by editing the object itself in a 3D editor. This can be solved by simply reducing the number of polygons or by using normal maps. Within the game engine, there is also an algorithm for optimizing 3D models, called the level of detail or simply LOD. The principle of this mechanism is to create several variants of the same mesh with different detail, and use them relatively to the distance from camera.

Figure 2. LOD example: 1148 polygons on the left and 147 on the right

Another important factor is the number of draw calls. Sometimes, a large number of small elements, such as decorations or plants, can decrease the performance of a game application. One possible solution is to this problem is

to combine these objects into one, which will allow you to render an entire group of objects with a single call to the graphical API. However, this technique can have negative consequences. The fact is that the merging of objects involves the merging of their materials as well. As a result, material processing requires more time and resources [Unreal Academy, 2019].

Peculiarities of game development

A game engine optimizer, like any other software product, is best created using an IDE. The choice of IDE depends entirely on the development platform, supported programming languages and the personal preferences of a developer. In the case when there are several languages in the project (e.g. C++ for the infrastructure and Python for scripting), the best solution would be a combination of two development tools: one for the main programming language, and the other for writing scripts.

Testing an optimization tool requires interaction with a supported game engine. Testing can be automated or done manually. In case of automating the testing a program logic, it is obvious to use unit testing libraries (e.g. Google Test) that easily deals with such problem. However, when it comes to UI, animation and other complex processes, developers use tools for automated testing (e.g. Gauntlet Automation Framework).

Using of several programming languages is a common phenomenon in game development. It allows developers to speed up the workflow by writing scripts in languages with a high level of abstraction (e.g. Python, Lua), while maintaining high performance using languages with a low level of abstraction (e.g. C++). Connection of these layers of a project requires using of a specialized build tool that will merge all scripts and core logic into a fully-fledged software product.

Being the main language for developing games, C++ is often accompanied by other languages used for scripting or other particular problems. Lua is a common choice due to its high performance and remarkable interoperability. C# is another popular language that is used in game development. While using it as a scripting language, C# still remains a powerful programming languages with a lot of tools for code maintenance. Python is also used as a C++ partner

for scripting tasks due to its high popularity, large open source code base and extreme ease of use. However, existing languages have numerous constraints and caveats, and, to avoid them, companies can create their own language from scratch or on top of existing ones.

Rendering optimization approach

First of all, a developer should explore the characteristics of the engine for which the optimization tool is being created. The developer should study its structure, the capabilities of its editor, the approach to creating plugins, methods for adjusting post-processing effects and available graphics profiles. Based on the gathered information, the developer can determine strong and weak points of the engine, what can be optimized, and what should be optimized.

During the design phase, it is necessary to consider the interaction mechanism of the tool with the engine, to create the prototype of the interface, to choose programming languages, development and testing tools. Even being a programming tool with complex algorithms inside, the optimization application its main target audience are designers and beginner developers, therefore the UX is vital for achieving a high download rate and getting relevant feedbacks.

Writing the code can be done in three stages: backbone, beta version and release. At the first stage, the developer gets acquainted with the mechanism of interaction with the game engine and available tools, creates the bare minimum for testing and the core infrastructure for the future functionality. The next step involves the implementation of all necessary functional requirements. After conducting the necessary testing and receiving feedbacks, developers are required to add new functionality (if such a decision was made) and bring the tool to the desired state. Typically, the interface design is also edited at this stage to make it more user-friendly.

The finished tool is recommended to be shipped using engine-specific marketplaces. The maintenance of the tool depends on user feedback, and it may be decided to add new functionality, change the existing one or add support for new tools.

Requirement analysis of the proposed approach

As the tool is designed for game developers with the lack of experience with the optimization mechanisms of the game engine, it must solve a range of specific problems. The goals of the tool are:

1. Facilitate the use of the game engine editor functions;
2. Accelerate the process of developing game projects;
3. Simplify the rendering optimization process in game projects.

To meet all business requirements, the developer must add a significant amount of functionality for the tool. The list of functional requirements should contain the following items:

- Ability to open the tool menu;
- Ability to create LOD for multiple objects at once;
- Ability to change light from dynamic to static, and vice versa;
- Display of the number of processed frames per second;
- Estimation of time costs for each category of graphic processes;
- Ability to change the post-processing effects for multiple areas;
- Work with many light objects at the same time.

Creating LOD is one of the key processes in optimizing rendering. Most game engines require a developer to create and configure them yourself. The tool should make this process easier and faster.

Dynamic light is a very expensive graphics process, as it checks the coordinates of an object and creates shadows in each frame. In many situations, it is safe to replace dynamic light with static light, which does not generate shadows in real time. Static light redraws the textures of the objects around at compile time, so it has almost zero performance overhead.

The post-processing effects also significantly slow down the rendering process. Effects such as ambient occlusion, depth of field, and bloom are often used, therefore the tool should control all of them.

To monitor the performance of the tool, user need to obtain information about the rendering process. That being said, it is necessary to include both the FPS counter and the GPU profiler into the tool.

System requirements are the same as of the targeted game engine. Crucial parts of the hardware are GPU, CPU and RAM, therefore one should consider using best parts available. Recommended specifications are:

- GPU: Nvidia GeForce 2080 RTX;
- CPU: AMD Ryzen 9 3900X;
- RAM: 32 GB.

Peculiarities of the proposed approach

Standard rendering procedure is described on the figure 3.

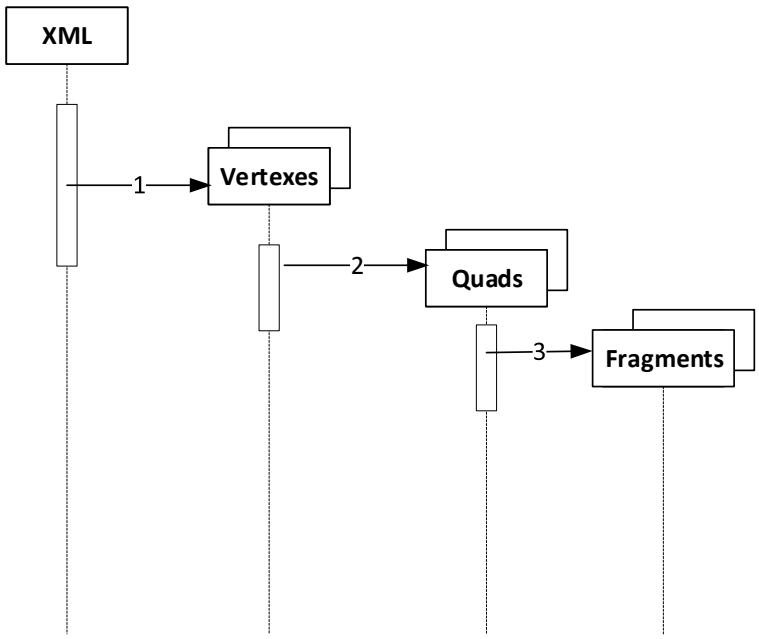

Figure 3. standard rendering procedure

Represent explanation about sequence diagram messages.

Model is stored in XML file. The first procedure (message 1) is to obtain vertexes from it. Then a list of quads is generated (message 2). After that quads are divided into fragments to implement different shaders (message 3).

Proposed approach contains additional operations aimed to reduce rendering time and used resources for rendering.

Sequence diagram of the proposed approach is represented in the figure 4.

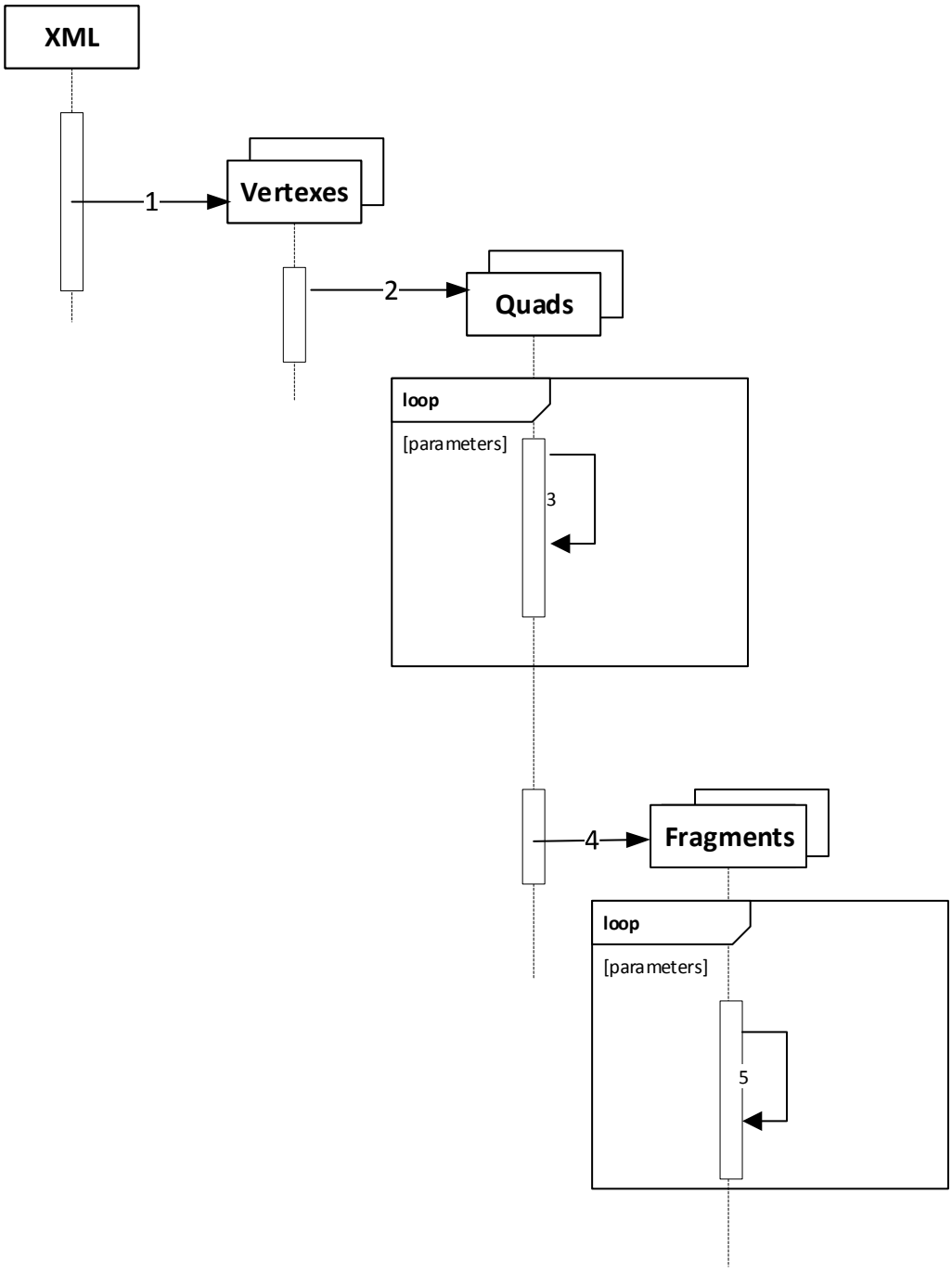

Figure 4. Proposed rendering approach.

Message 3 contains the set of operations aimed to optimize number of vertices in model. In our software tool user can manage LOD that influences to number of tessellated polygons in model. Message 5 aimed to optimize number of shaders aimed to provide realistic representation of rendered picture.

Analytical representation of rendering process

In order to propose flexible and extensible approach analytical form of optimized rendering procedure is performed. Foundations of analytical representation of UML behavioral models, using graph representation, are proposed in paper (Chebanyuk, 2018a).

Sequence diagram (f)contains more complex structure in comparison with structures possible to consider in that paper. Namely, it has cycle constructions, needed to be expressed with parameters.

That's why we consider the continuation of this approach, proposed in (Chebanyuk et. al.,2018) where analytical representation of cycle operation are proposed

$$O = ((el_1, l_1, el_2), ..., (el_n, l_m, el_k), (el_k, l_1, el_p), ...(el_t, l_r, el_w), (el_w, l_s, el_1)) \quad (1)$$

where O – is a denotation of cycle operation.

Analytical representation of general rendering procedure is the next:

Compose an array of objects and links for the UML Sequence Diagram.

$$\begin{aligned} O &= \{el_1 = XML, el_2 = Vertexes, el_3 = Quads, el_4 = Fragments\} \\ L &= \{1,2,3,4,5\} \end{aligned} \quad (2)$$

Representation of two loop operations.

$$\begin{aligned} O_1 &= \{(Quads, 3, Quads)\} \\ O_2 &= \{(Fragments, 4, Fragments)\} \end{aligned} \quad (3)$$

Representation of full chain of events

$$ch_1 - \{XML, 1, Vertexes\}, \{Vertexes, 2, Quads\}, O_1, O_2$$
$$O_1 = \{(Quads, 3, Quads)\} \quad (4)$$
$$O_2 = \{(Fragments, 4, Fragments)\}$$

Such representation allow to understand general idea of rendering optimization approaches. Cycle (3) can include necessary number of operations setting polygon parameters. Cycle (4) can include necessary number of operation aimed to optimize fragment representation, as it is needed for the best realization by architecture, operating memory, and other resources included to rendering. Taking 0 operation one can skip one cycle and add as many operations as it needed to other cycle.

Experimental results

The user interface is presented in the form of a menu that contains all necessary functionality to interact with the optimization tool. The menu is presented in the form of modules, each for a separate category of optimization settings. The module contains a brief description of the functionality inside. All functions has a short description and a button to call them. Additional options are possible for some of them. The prototype of the module is shown in the Figure below.

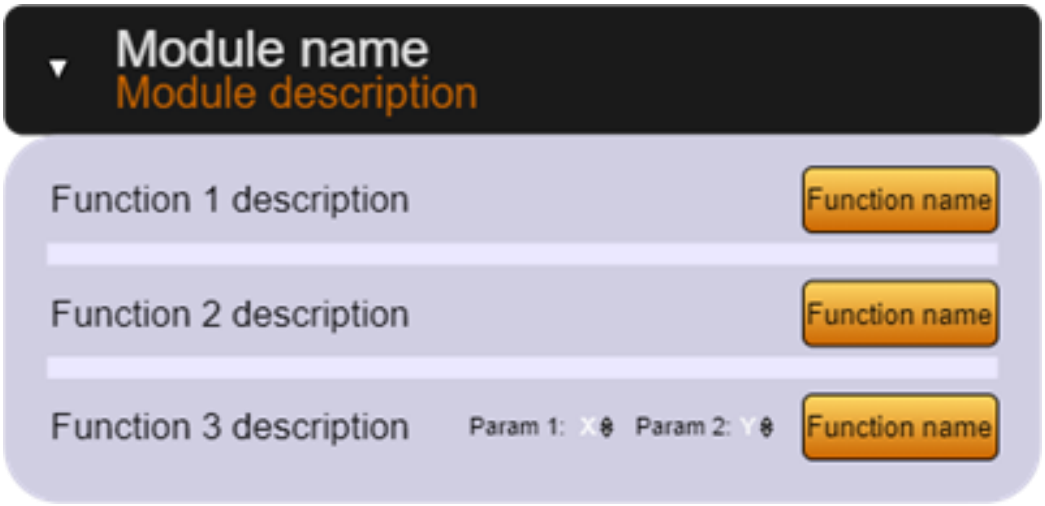

Figure 5. Main menu module interface prototype

The top panel of a module contains a name, a short description and a button that can hide/reveal the functions of the module. Each function is placed on a separate line. There is a brief description of the function actions on the left side of a line, and a button that calls the function on the right. Also, to the left of the button, there can be parameters which a user can set manually if such functionality is provided.

Functions can either start certain procedures for processing game scene objects or open submenus to manage settings. The submenu is presented as a list of parameters that can a user can assigned a value or turned on/off. The prototype of the submenu is shown in the next Figure.

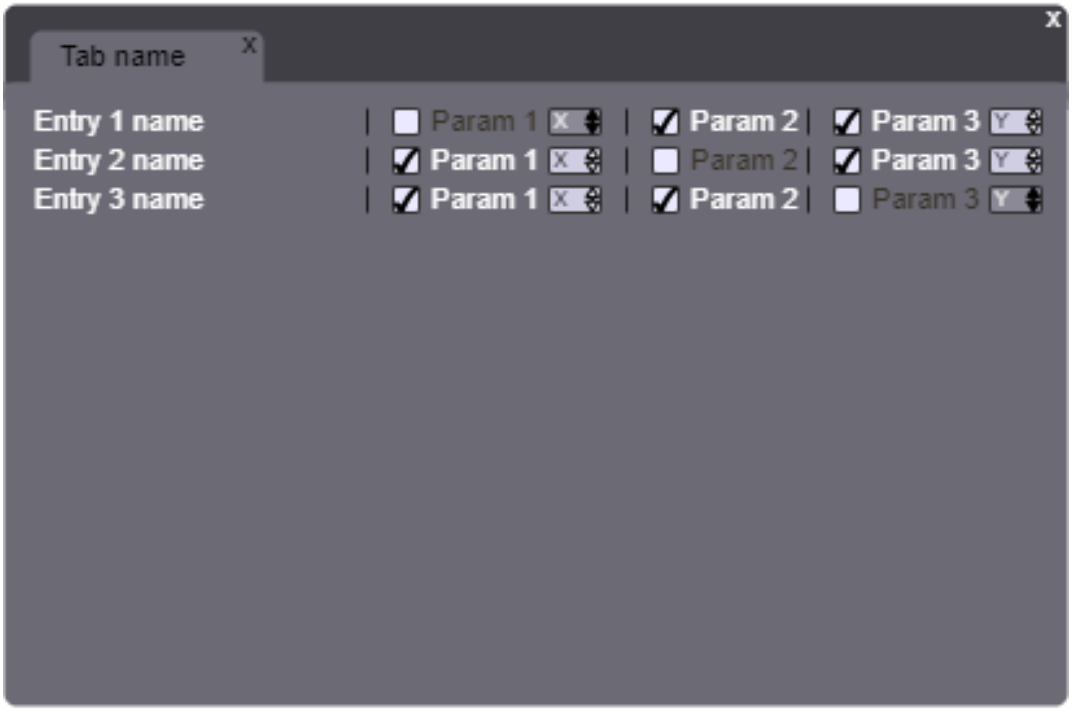

Figure 6. Submenu interface prototype

A submenu opens in a separate tab, which by default is displayed in a new window. Each object is located on a separate line. There is the name of an object on the left of the line, and the parameters on the right side. Enabled settings are highlighted when disabled are darkened. Also, if the setting is disabled, its value cannot be edited. This is indicated by the dimming of the parameter selection field.

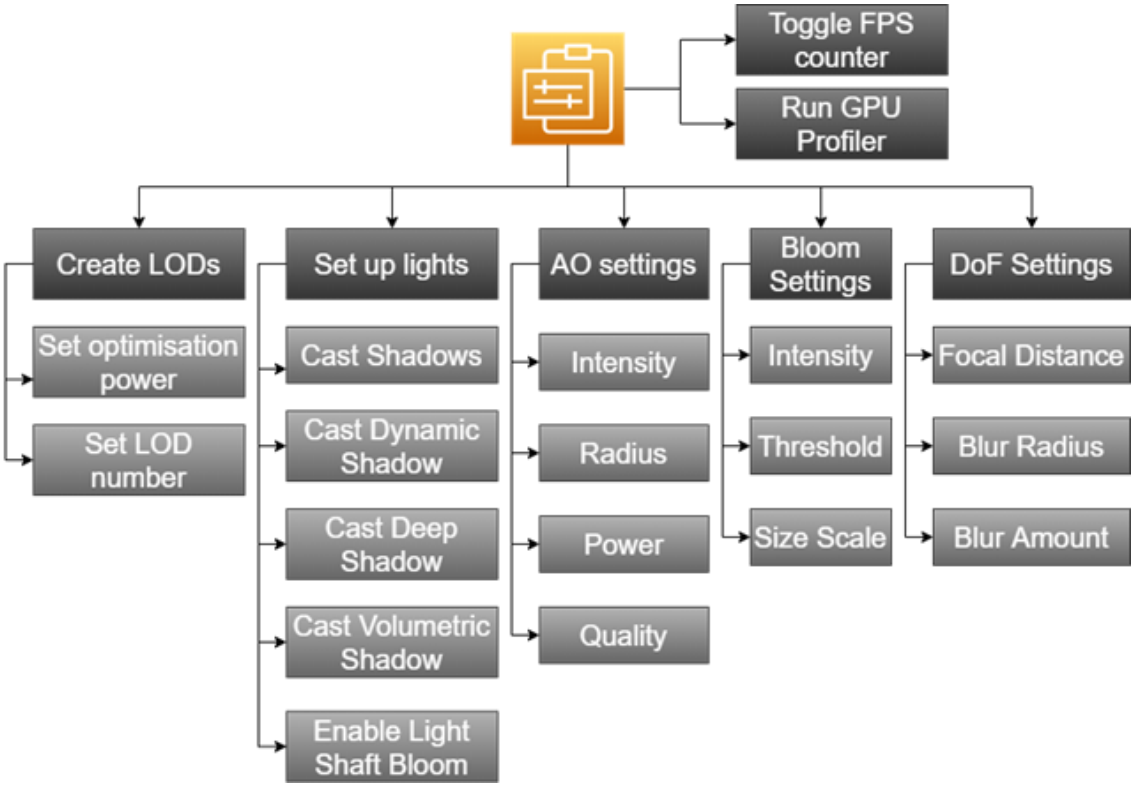

Figure 7. Tool functions and their parameters

The Create LODs function allows a user to create LOD for each selected object that belongs to the Static Mesh Actor class. The user can specify the number of LODs for this group of objects, as well as the power of optimization of static meshes. The optimization power is the coefficient by which the number of polygons for the next LOD will be determined. For example, for a static mesh of 1000 polygons, the user selected the number of LOD 3 and set the optimization power to 2. In this case, the tool will create 3 LODs with the following number of polygons:

- LOD 0: 1000 polygons;
- LOD 1: 500 polygons;
- LOD 2: 250 polygons.

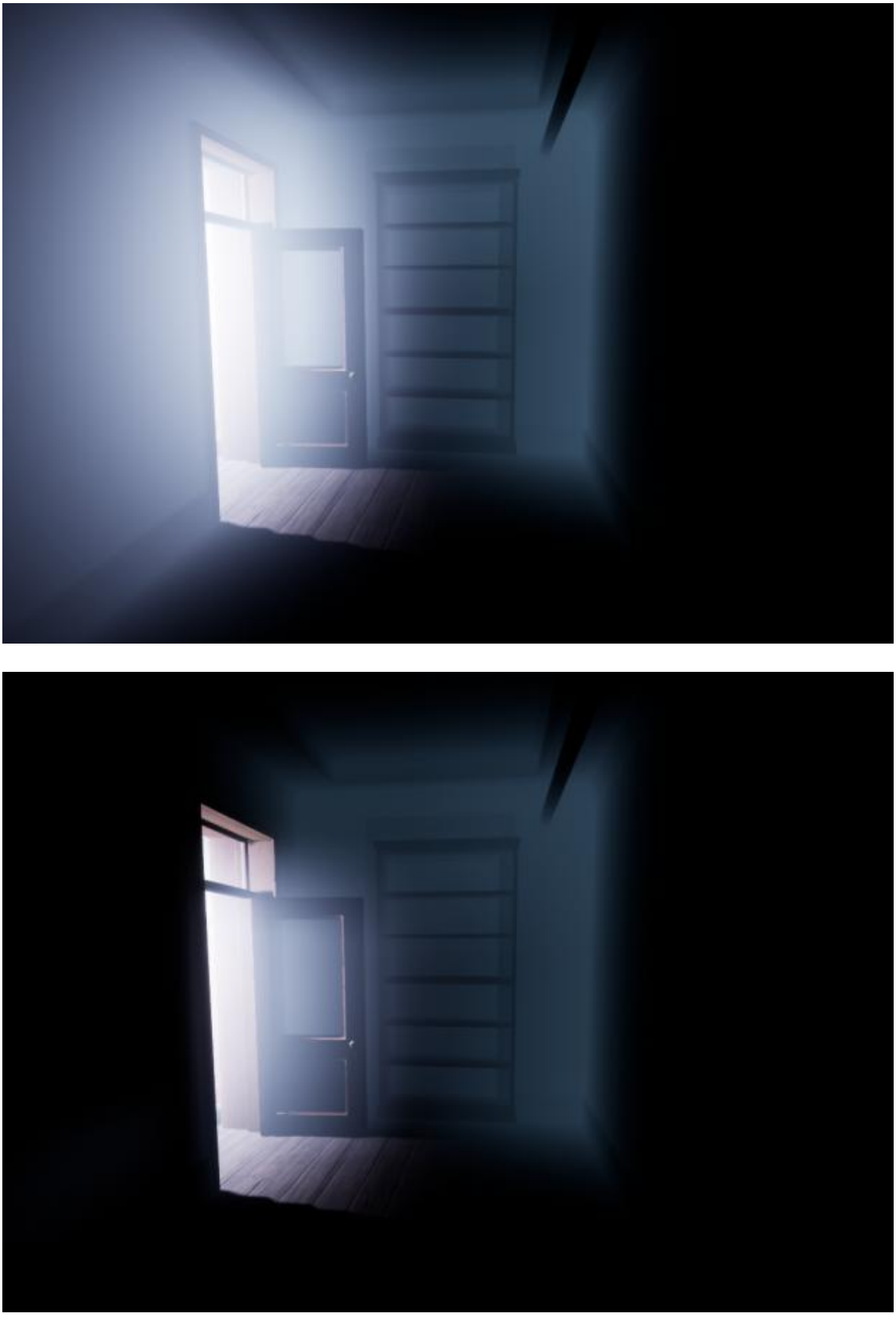

Figure 8. Cast Volumetric Shadow turned off (left) and on (right)

The Set-up lights function allows user to adjust all the light objects at once that are on the game stage. This is achieved by displaying these settings in a separate submenu. A brief explanation of the parameters is given:

- Cast Shadows determines whether a shadow will be generated from this light object;
- Cast Dynamic Shadow determines whether the light object will generate a shadow in real time;
- Cast Deep Shadow determines the quality of the shadow;
- Cast Volumetric Shadow allows you to enable or disable the shadow scattering effect;
- Enable Light Shaft Bloom determines whether the rays of the light object will create a glow effect.

The AO Settings function allows the user to adjust the ambient occlusion effect. This feature is also displayed in a separate submenu to allow editing of multiple post-processing areas at once. The parameters of this function are the intensity setting of the ambient occlusion effect, the radius of propagation, the strength and the quality of the ambient occlusion.

The Bloom settings function allows the user to adjust the glow effect. The principle of operation is similar to AO Settings, only with other parameters. Here the user can assign values to the intensity of the bloom effect, the maximum threshold and the coefficient by which the levels of the bloom will be determined.

The DOF settings function allows you to adjust the depth of field effect. It works similarly to the two previous functions, but the parameters here are focal distance, blur radius and blur amount.

The last two functions allow you to get information about the rendering process. The first (Toggle FPS counter) allows to get the number of frames processed per second. The second (Run GPU Profiler) allows to get the information about the time spent on each of rendering processes.

Figure 9. Comparison of minimum ambient occlusion power (left) and maximum (right)

Figure 10. Intensity of a bloom effect is set to minimum (left) and maximum (right) value

Business logic

Business logic functions are mostly focused on implementing the functionality available through the user interface. However, a special software infrastructure is required for the tool to function properly.

One of the most important infrastructural solutions of business logic is to make a widget in a separate window (Dispatch Widget). This is achieved by calling the procedure for creating a tab in the editor and assigning a widget to it. This feature does not create a widget on its own to allow the user to modify the widget before tabbing.

Unreal Engine does not allow to make changes to the post-processing area object [Unreal Engine 4 Documentation, a]. To provide settings through the tool menu, developer needs to create a number of accessors and mutators of the fields of the post-processing settings class.

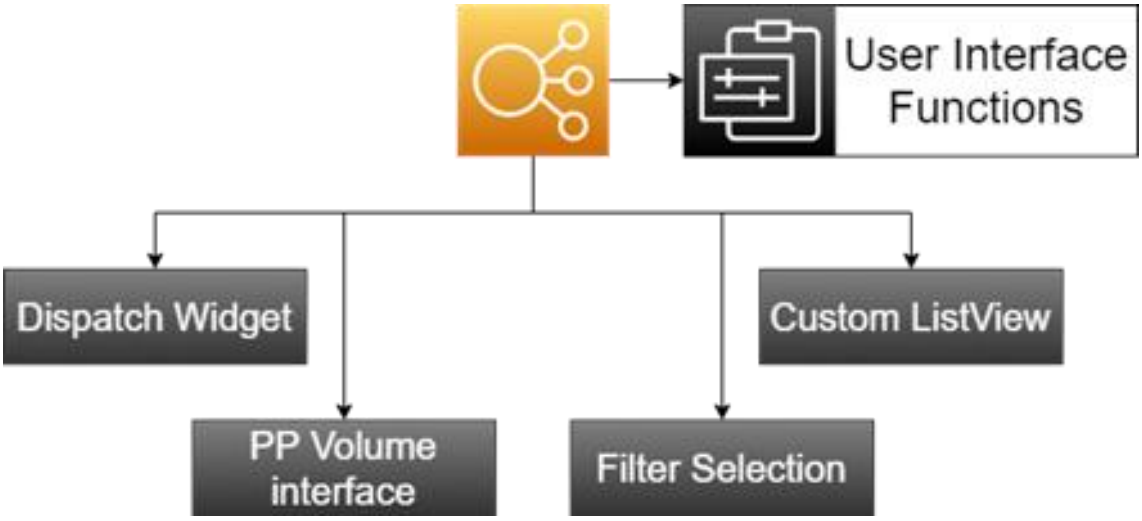

Figure 11. Business logic infrastructure

Another problem with Unreal Engine is that it does not allow changing the ListView's internal widget class from the outside [Unreal Engine 4 Documentation, b]. At the business logic level, this was solved by extending the ListView with the addition of a mutator to change the internal class.

To ensure the safe creation of LOD without accidental errors in the form of an attempt to access a static mesh of a class other than Static Mesh Actor, a filtering function was created. The input parameter is an array of any objects from the game scene, and the output is an already filtered array containing only the Static Meshes Actor classes.

Implementation

The menu consists of two modules: general settings and post-processing settings. General settings include functions for displaying the FPS counter, launching the GPU profiler, adjusting light objects, and creating LODs. The post-processing settings includes ambient occlusion settings, depth of field settings, and bloom settings.

C ++ language was chosen to write the infrastructure of the tool, and the logic of the user interface was created using the Blueprint visual programming technology.

The infrastructure includes two blueprint function libraries and a UListView extension class. The first library contains all the important accessors and mutators for working with the post-processing settings of the APostProcessVolume class, and the second library contains all other editor extensions. The functions of these libraries are not private, so the user can use them outside the menu of the rendering optimizer.

A number of widgets have been created for the interface, and each of them is responsible for displaying their own specific category of settings. One of the most important widgets is the submenu widget which provides a template for creating list widgets. There is also a widget among them that serves as a template for menu modules. All other widgets are used in specific menu functions as internal widget classes for the ListView of the submenu widget.

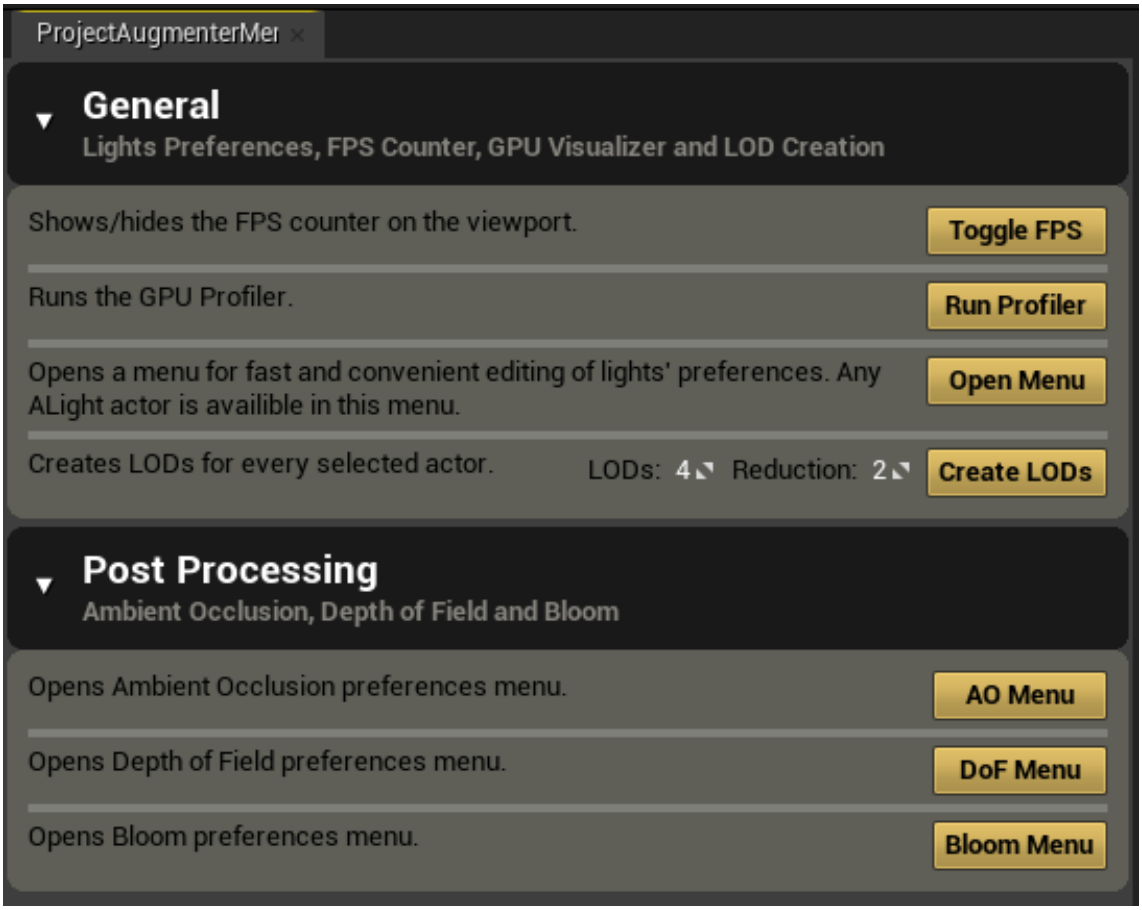

Figure 12. Finished tool main menu interface

Testing results

It was impossible to test the prototype in a complex project, however, even in primitive one it gives fascinating results. Adjusting the light and creating the LOD gave the biggest increase in performance. The testing was performed on devices with the following characteristics:

- OS: Windows 10 64-bit;
- CPU: Intel Core i5-7500 3.4 GHz 4 cores;
- RAM: 32 GB;
- GPU: Nvidia GeForce GTX 1060;
- DirectX version: DirectX 12;
- Unreal Engine version: 4.26.2.

Although it is impossible to estimate the true effectiveness of the tool on a primitive project, even in this case the tool was able to save 1.54 ms on each frame, which is 13.56% of the initial result.

Conclusion

Paper proposes approach to increase rendering speed due to flexible settings of procedures needed to optimize rendering process (1)-(5). Flexibility allows to user of game framework to choose set of settings advisable to archive the best representation of series for rendered frames from different kinds of settings (figures 8-10).

Testing results show raising of effectiveness for fixed hardware configuration.

In order to verify proposed analytical approach of rendering optimization requirements specification for the development tool for Unreal Engine is designed. Then the architectural solution is proposed.

The main purpose of the developed tool is to accelerate the development of game projects and facilitate the settings of graphics. In the future, it is planned to refine the tool and add more functionality needed for graphics settings.

In the future, this will allow to quickly and easily expand the functionality of the optimization tool.

Further researches

Further researches are aimed towards the improvement of the tool and spreading analytical approach of rendering optimization described in this article. It is planned to implement a project evaluation system that will give user suggestions and will show the mark of the project depending on the hardware bottleneck and current performance. The other important feature is the custom GPU profiler that will give more information about rendering processes and will allow to test multiple specified places in the virtual scene in one run.

Bibliography

(Chebanyuk O., 2018) An Approach of Text to Model Transformation of Software Models. In Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018), 432-439

(Chebanyuk O., et al, 2018) Chebanyuk O. & Dyshlevyy O. & Skalova V. An approach of obtaining initial information for behavioral software models processing. International journal "Informational Models and Analysis". Volume 7, Number 2, 2018, 25-41.

[Intel, 2017] Intel. Unreal* Engine 4 Optimization Tutorial, Part 4 . 2017. https://software.intel.com/content/www/us/en/develop/articles/unreal-engine-4-optimization-tutorial-part-4.html

[Unreal Academy, 2019] Unreal Academy. Optimization Challenge. 2019. https://learn.unrealengine.com/course/2547341/module/5525720?LPId=89346

[Unreal Engine 4 Documentation, a] Unreal Engine 4 Documentation. FPostProcessSettings. https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/Engine/FPostProcessSettings/

[Unreal Engine 4 Documentation, b] Unreal Engine 4 Documentation. List View. https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/ListView/

Authors' Information

Olena Chebanyuk *– Software Engineering Department, National Aviation University, Kyiv, Ukraine,*

Major Fields of Scientific Research: *Model-Driven Architecture, Model-Driven Development, Software architecture, Mobile development, Software development,*

e-mail: chebanyuk.elena@ithea.org , chebanyuk.elena@gmail.com

Mykhailo Mushynskyi *– National Aviation University, Kyiv, Ukraine; Software Engineering student,*

Major Fields of Scientific Research*: Game Development, Game Engines,*

e-mail: mmmikeuser@gmail.com