Restricted Extensions for GPU Photo-realistic Renderer

V.V. Sanzharov[1], V.A. Frolov[2,3], I.V. Pavlov[3]
vsan@protonmail.com|vova@frolov.pp.ru|ipa4lov@gmail.com
[1]Gubkin Russian State University of Oil and Gas, Moscow, Russia;
[2]Keldysh Institute of Applied Mathematics RAS, Moscow, Russia;
[3]Moscow State University, Moscow, Russia

Photo-realistic rendering systems on CPU traditionally have significant flexibility achieved mainly by the ability for end user to write custom plugins or shaders. The same cannot be said about majority of photo-realistic GPU renderers. Most «classic» approaches to design of user-extendable software on CPU, such as object-oriented plugins are not very well suited for GPU programming. In this paper we propose a restricted approach to developing extendable GPU rendering system at low development cost. Our hardware agnostic light-weight approach can be applied to existing rendering systems with minimal changes to them. We apply our approach to the problem of procedural textures implementation and show that, in addition to simplicity, our approach is faster then existing GPU solutions.

Keywords: *photo-realistic rendering, ray tracing, GPU, procedural textures*

1. Introduction

Modern photo-realistic rendering systems are mov-ing towards GPU implementation with many strong players actively developing GPU versions of their products [19]. This tendency is becoming even more pronounced with the advent of hardware accelerated ray tracing technology (i.e. Nvidia RTX) which is available to general public. However, for a long time industry (such as visual effects, animated films, architectural visualization and others) used CPU renderers which are known for their flexibility and extensibility. By these terms we mean the ability given to the end-user of the rendering system to easily add new features such as procedural shading, texturing, custom BSDF or light source models.

1.1 Related work

1.1.1 Plugins

One of the most powerful traditional approaches is object-oriented plugins [14, 18]. This approach is highly flexible, however it has well-known drawbacks and limitations. One of the most serious problems — inability to use hardware at full speed due to encapsulation [1]. For example, SIMD will require to change interface and even then, we have to sacrifice portabil-ity. Although the object-oriented approach is possi-ble on the GPU, its efficiency is extremely low due to GPUs are not designed for dynamic-dispatching code [2]. Another problem is the actual limited flexibility of the interfaces: it is impossible to create an interface that will satisfy all users in the future. Thus, Do-main Specific Languages (DSL) is the more powerful approach.

1.1.2 Domain Specific Languages

One of the first methods to formulate custom shading operations was «shading trees» [3] - directed acyclic graphs with input values in leafs, operations (such as addition and multiplication) in nodes and final value in the root. Different materials, lights, atmospheric effects are formulated with different trees which are combined by a specific procedure by rendering system. Such trees can't specify loops or conditional execution. In pixel stream editor proposed in [13], custom procedure is executed on every pixel using some arbitrary per-pixel data as input. These two approaches served as a basis for RSL. In RSL there are shaders of different types (light, surface, volume, etc.), which are called by the rendering system. Computations can be executed with different rates — per-batch and per-sample.

One of the first shading languages - RSL (RenderMan Shading Language) was developed as a part of RenderMan [7] and more modern Open Shading Language (OSL) [27] was initially developed for Arnold renderer. Both of these rendering systems are currently CPU based, although GPU version of Arnold is in development. Open Shading Language (OSL)[27] was specifically designed for ray-tracing based algorithms. It uses LLVM framework and just in time (JIT) compilation. The shaders are first compiled to bytecode and then translated into x86/x64 instructions. Approaches based on full-fledged compilers are certainly the most powerful and flexible. Their main disadvantages are high complexity, high development cost and obstructed debugging. In any case, exist-ing solutions uses CPU: RSL [7], OSL [27], VEX [22] and other [4]. Although, Octane GPU renderer implements a subset of OSL for procedural textures [26].

GPU programming enforces more restrictions and is more difficult in general, so it is hard to design comparable flexible solution with efficient GPU implementation which will preserve high performance. In real-time graphics applications the difficulty is mitigated by the fact that there exists standardized graph-ics pipeline with known stages. In ray tracing sim-ilar «pipeline-like» approach was adapted in OptiX [11], RTX [25] and OpenRL [21]. However, it requires the whole rendering system infrastructure to be im-

plemented using these technologies which are limited to specific hardware. Running ahead, our approach is hardware-agnostic.

There are several well-known hardware-accelerated shading languages: GLSL, HLSL, Cg, Metal. There are approaches that propose new languages compiled to one or several hardware shading languages such as [15] to improve portability across different rendering engines. In [8] authors propose approach to building shaders from modular components written in a do-main specific language. In shading language proposed in [6] shaders are mapped to more than one graph-ics pipeline stage and implement the concept of in-heritance from object-oriented programming to make shaders easily extendable and reusable.

In [10] a framework suitable for creation of cus-tom shader pipelines is proposed. These pipelines are mapped to a series of kernels which are then sched-uled for sequential execution. Pipelines can target different hardware such as GPUs or multi-core CPUs. Authors demonstrate application to rasterization and Reyes pipelines. Authors in [12] propose Ray Tracing Shading Language (RTSL) which is based on GLSL and to some extent on RSL. RTSL was developed with CPU rendering systems in mind and makes use of SIMD extensions and packet tracing.

Another solution is OptiX ray-tracing engine by Nvidia [11]. It provides the user with the ability to create programs of different types - an approach simi-lar to conventional graphics APIs. OptiX can be hard-ware accelerated on Nvidia Turing hardware [24] by the means of RTX technology. RTX is also available in the other graphics APIs (DirectX and Vulkan) and an approach largely similar to OptiX can be used with these APIs instead [25]. Caustic Graphics OpenRL [21] was the earlier technological analog of same ideas with hardware acceleration which did not reach mass product scale.

1.1.3 Disadvantages of existing GPU approaches

OptiX [11] utilizes *mega-kernel execution model* where user programs are linked together into a mono-lithic kernel. It behaves like a state machine where the state identifier selects what program should be executed next with some optimizations to reduce reg-ister pressure (by using same register set for differ-ent stages). This approach is considered ineffective because of several reasons [9], [5]. First, it uses sin-gle, maximum register number for of all programs and some states degrade performance of others in this way. Second, it is complex to profile and has unstable per-formance. Finally, high branch divergence between states can significantly degrade performance. To be fair, it should be said that this approach still has ad-vantages. First, no overhead for launching kernels (which presents in other approaches) because all code is merged into a single kernel. Second, this approach allows recursion support, function calls and exceptions [11].

An alternative to *mega-kernel* is simple divide and conquer strategy, called *separate-kernel* [5]. This ap-proach avoids high register pressure via manual split-ting of code in to several kernels. Unfortunately, it can't help with extension of renderer with custom user programs by itself because *separate-kernel* is manual optimization by it's definition.

In [9] *wavefront* pathtracing was suggested. The idea of wavefront pathtracing is to push and exe-cute different stages of *mega-kernel* state machine in *separate GPU queues*. Wavefront pathtracing solves both branch divergence and register pressure problem as different programs are actually executed in sepa-rate kernels. This is true for both user and render-system defined programs, thus, ray-sorting/thread-sorting approaches are enabled here. Wavefront ap-proach also has limited support for recursion as it can be thought of as breadth-first search in tree. It seems that Nvidia RTX partially uses this approach (at least for ray-scene intersection). Unfortunately wavefront pathtracing also has disadvantages. First, it has unpredictable memory footprint and kernel ex-ecution overhead in general. This is mainly because each «call of a function» that actually executes in sep-arate queue must perform at least several operations:(1) append all arguments of «a function» to global memory, (2) save the whole current state of execut-ing program in global memory, (3) read current state of executing program from global memory (with the returned result) and continue executing when child queue of «a function» will be completed. With the addition of recursion this approach quickly becomes memory-hungry due to high breadth-first search cost combined with large amount of threads (100K–1M). Second, wavefront approach has limited ability to par-allelize computations because user-defined procedures are not guaranteed to create work in parallel. And usually they do not. Imagine a common case of shoot-ing several rays in RTX samples via loop that leaves no other choice for implementation other than to pro-cess all rays in groups for the first iteration, second and subsequent iteration of loop. Such approach sim-plifies things for user but limits efficiency.

Thus, *mega-kernel* is more natural for CPU and *wavefront* approach is better for GPUs. Neverthe-less, both mega-kernel and wavefront methods have enormous implementation cost, huge complexity and obstructed debugging for end-user as the final gener-ated code and execution model are greatly different from original input code. This approach is suitable for large companies that heavily invest in compilers and hardware.

1.1.4 Closest analogues

When using existing solutions small development teams become dependent on hardware and lose the

portability of their products. At the same time they have no resources for supporting their own compilers and thus often prefer not to use these technologies at all [20, 26]. Both Cycles [20] (open source) and Octane[26] (commercial) restrict their programmabil-ity to procedural textures. Shaders can evaluate input parameters for material and light models, but can not change these models.

There are several reasons for such decision. First, programming/extending things like material or light source models is too hard for end-user. This is mainly due to the fact that modern rendering systems use advanced light transport algorithms with multiple im-portance sampling and each material and light should be able to not only generate samples but also calcu-late their probability density (both forward and re-verse if BPT[17] is used for example) which is a quite tricky task — correctness of light integration could be easily broken by the user. Second, excessive exten-sibility leads to re-compilation for the whole system every time (OptiX approach) and can be too long and inconvenient for the end-user. Thus «shading trees» approach is still widely used for extending materials. Moreover, shading trees are convenient for artists who use visual programming approach via some GUI to simply «draw» them.

Cycles approach is mature but has disadvantages. Cycles transforms acyclic shader graph directly to GLSL code which is further compiled in to a kernel. It's first disadvantage is that the actual argument val-ues mapping is produced during the code generation process. So, the input values are placed in GLSL code as constants and if some procedural textures are used twice with different argument values Cycles will dupli-cate calls (which will definitely lead to branch diver-gence for different rays executing same texture with different parameters). Apart from the fact that the effectiveness of such a decision is questionable, any change in the parameters of procedural textures will lead to a recompilation of the final texture kernel each render launch. Second, Cycles does not allow to add new basic nodes «on the fly». It requires that all ba-sic nodes to be described in a single place and to be free of name conflicts. Adding new basic node leads to the whole system (Cycles) recompilation. Finally, generated code is complex for debugging (in any way) due to aggressive code generation.

2. Proposed solution

Our render system is designed in separate-kernel approach [5] (fig. 1). In analogue to Cycles we allow for programmable procedural textures only to restrict complexity and preserve performance for «fixed» func-tionality of the system. This way we achieve true modular architecture where it's easy to change the particular part of ray tracing algorithm (for example acceleration structure traversal or BRDF evaluation) without affecting the others.

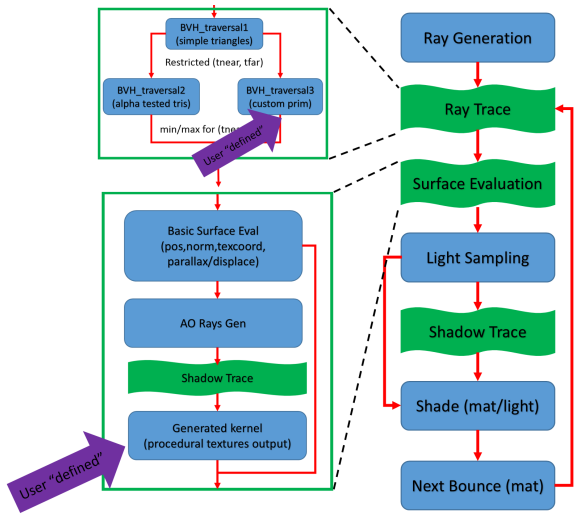

Fig. 1. Our architecture. Generated kernel is shown with an arrow. Users do not write this kernel directly. Kernel is generated in a certain way from a set of user defined procedures.

It is also easy to add new functionality by intro-ducing new kernel as additional computational step. This is the way we implement procedural textures —we add new kernel which executes procedural texture code on hit and writes the results into the global mem-ory, other stages are not affected. With this approach the task of integrating procedural textures code sub-mitted by the end-user into the renderer is reduced to development of some mechanism to properly «in-sert» it into procedural textures kernel (as in Cycles). However, this is where our similarity ends.

Unlike Cycles we allocate a separate memory area inside material buffer for storing argument values there (fig. 2). We call this area «fake stack». The code generator can think of this memory region as a stack, placing arguments in it. But since all parame-ters are constant, «fake stack» in fact is just a read-only region of global memory. Thus, we can update this region from the CPU side without kernel recom-pilation. Besides, «fake stack» region is almost unlim-ited in space so (unlike Cycles) we are not limited in amount and size of arguments. Thus, the assignment of the actual value of the argument occurs when the texture is attached to the material.

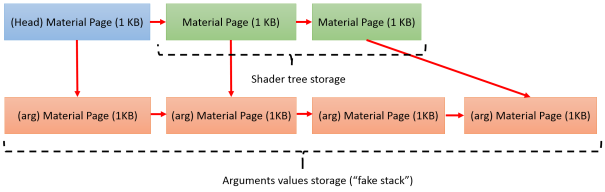

Fig. 2. Material layout in memory. Blue and green material pages (first three rectangles) represent different BRDF nodes (of shader tree) that may reference different regions of argument storage in the same buffer.

The next major difference of our approach and existing (at least Cycles) is nested procedural textures processing. Existing approaches usually just inline code of child textures in to the parent. This approach works well if one construct the final nested texture from a large number of small base nodes. However for heavy nodes (like Perlin or other noises which are quite common for procedural textures) this leads to excessive waste of registers due to heavy code duplicated several times.

Our solution comes from assumption that basic building blocks (defined by user) are heavy enough in their majority. For such case it is more efficient to process them in the same way as interpreter does (fig. 3). This can be thought of as a restricted *mega-kernel* approach. We allocate small stack inside kernel and use DFS traversal of texture graph to get topo-logical order. This would help us to ensure that all parameters needed for this function call have already been calculated before the call. Results of intermedi-ate calls are also stored on stack because they could be used as a parameters in next function calls.

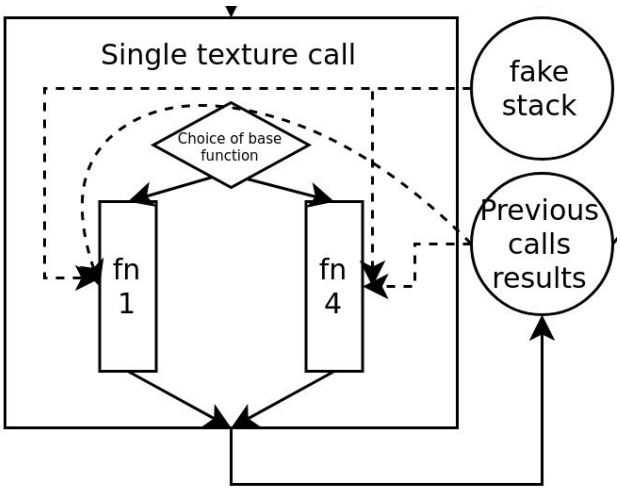

Fig. 3. Interpreter of nested textures. A restricted mega-kernel approach

At last, we implement some custom features like ambient occlusion (AO) in «fixed function» by introducing new kernel that computes **all AO rays in parallel**. This gains us additional performance over naive approach (table 1).

2.1 Implementation details

We provide possibility to end-users to write their own custom procedural textures in OpenCL C99. To resolve name conflicts on different procedural textures we transform function names using Clang LibTooling. It provides possibility to build AST for OpenCL source code and then modify original source code using this tree. We add unique prefixes to all user functions calls, definitions and declarations according to the actual texture ids. We also used Clang to replace «embedded calls» and «embedded types» that allow procedural texture to read surface attributes («readAttr»), global engine settings and sampling from 2D images («tex2D») inside user code:

```
float3 pos  = readAttr("WorldPos");
float3 norm = readAttr("Normal");
float3 tang = readAttr("Tangent");
```

This approach separates implementation details from user and allow us to silently change implementation in future. Using Clang for code instrumenta-tion allows us to not agree with the user in detail about some kind of concrete interface. We intention-ally didn't use any new syntax so that the input user code is still 100% C99 code. In this way it can be eas-ily integrated with separate C/C++ application for debugging/testing or any other purpose. Example of simple user procedural texture which multiplies colors of 2 different images:

```
float4 userProc(sam2D tex1, sam2D tex2)
{
  float2 texCoord  = readAttr("TexCoord0");
  float4 texColor1 = tex2D(tex1, texCoord);
  float4 texColor2 = tex2D(tex2, texCoord);
  return texColor1*texColor2;
}
```

It should be mentioned that unlike existing approaches, our instrumentation changes code only slightly. This way it is possible to look at the generated kernel, directly modify and debug it the same way as any other OpenCL kernel. This is essentially different from Cycles approach. Also, it is possible to use the proposed approach to implement other extensions (not just procedural textures) to the rendering system such as procedural geometry.

2.2 Comparison

We compared our approach with RTX-based path tracer implemented in Vulkan API and simulated Cy-cles approach — we inserted all constants and textures directly in code of a single procedural textures to sim-ulate Cycles. As our test scenario we used Sponza scene with inserted 3d model. First we rendered the scene with simple diffuse light gray material on the model (produced images were substantially identical) and then changed material to use procedural texture (fig. 4). We measured the drop in performance to see how well relatively heavy computations in shading will be handled. Procedural rust is a blend between two materials - simple diffuse gray and the base rust mate-rial. The base rust material is based on noise functions and color ramps (fig. 5). The mask for blending in addition to noise functions also uses ambient occlu-sion. Scratches material is based on noise functions and voronoi pattern (fig. 6). Comparison results are shown in (tab.1).

Fig. 4. Scene with simple diffuse material on 3d model (left) and procedural texture (right).

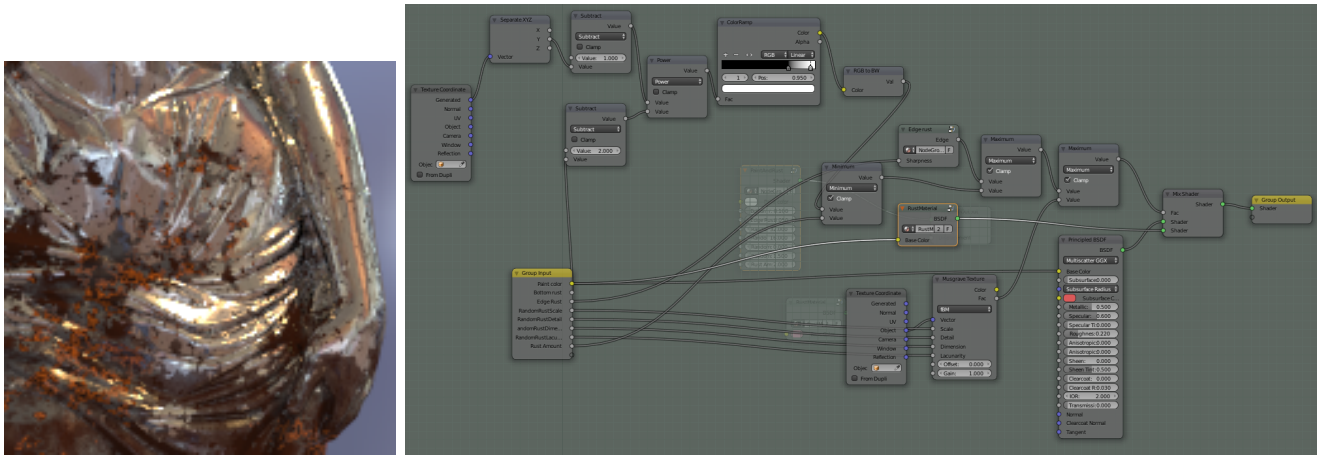

Fig. 5. Example of rust procedural texture applied to a model close-up (left) and shader node tree in Blender (right, some nodes are actually collapsed parts of the tree). Note, that different model and base material (with reflective component) are used here for the sake of image clarity.

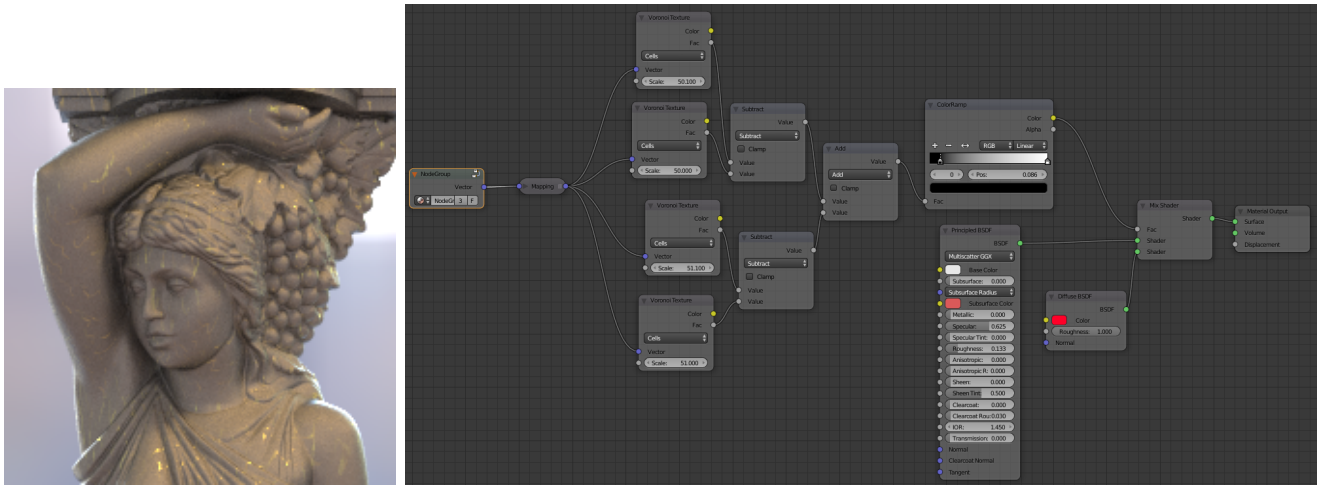

Fig. 6. Example of scratches procedural texture applied to a model close-up (left) and shader node tree in Blender (right, leftmost node is actually a collapsed part of the tree). Note, that different model and base material (with reflective component) are used here for the sake of image clarity.

procedural texture	Cycles sim.	RTX	**Ours**
rust (time)	+46%	+33%	**+21%**
scratches (time)	+9%	+4%	**+15%**
rust (stack frame)	604 bytes	unknown	**464 bytes**
scratches (stack frame)	412 bytes	unknown	**336 bytes**

Table 1. Rendering time increase with proc. texture compared to simple diffuse material and stack frame size obtained with "-cl-nv-verbose" key for OpenCL compiler.

In this way (table 1) using restricted mega-kernel we outperform Cycles approach for complex procedural textures (with ambient occlusion rays) in both speed and stack-frame size but have little loss of speed for simpler texture (scratches). We also beat RTX implementation for rust because proposed implementa-tion processes ambient occlusion rays in parallel while both RTX and Cycles are limited to tracing one ray at a time for each thread. In comparison with Cycles ap-proach we have smaller stack frame size for both tex-tures. This demonstrates that restricted mega-kernel approach performs its functions.

3. Acknowledgments

This work was sponsored by RFBR 18-31-20032 grant.

4. References

[1] Albrecht T. *Pitfalls of Object Oriented Programming.* Sony Computer Entertainment Europe Research and Development Division archives. 2013.

[2] Barik R. et al. *Efficient mapping of irregular C++ applications to integrated GPUs* //IEEE/ACM International Symposium on Code Generation and Optimization. – ACM, 2014. – p. 33.

[3] Cook R. L. *Shade trees* //ACM Siggraph Computer Graphics. – 1984. – Vol. 18. – №. 3. – p. 223-231.

[4] Deryabin N. B., Zhdanov D. D., Sokolov V. G. *Embedding the script language into optical simulation software* // Programming and Computer Software. 2017, Vol. 43, №1, pp 13–23.

[5] Frolov V., Kharlamov A., Ignatenko A. *Biased Global Illumination via Irradiance Caching and Adaptive Path Tracing on GPUs.* GraphiCon'2010, p. 49-56.

[6] Foley T., Hanrahan P. Spark: modular, composable shaders for graphics hardware. – ACM, 2011. – Vol. 30. – №. 4. – p. 107.

[7] Hanrahan P., Lawson J. *A language for shading and lighting calculations* //ACM SIGGRAPH. – ACM, 1990. – . 24. – №. 4. – p. 289-298.

[8] He Y. et al. *Shader components: modular and high performance shader development* //ACM Transactions on Graphics. – 2017. – Vol. 36. – №. 4. – p. 100.

[9] Laine S., Karras T., Aila T. *Megakernels considered harmful: wavefront path tracing on GPUs* //HPG'13. – ACM, 2013. – . 137-143.

[10] Patney A. et al. Piko: a framework for authoring programmable graphics pipelines //ACM Transactions on Graphics. – 2015. – . 34. – №. 4. – p. 147.

[11] Parker S. G. et al. GPU ray tracing //Communications of the ACM. – 2013. – Vol. 56. – №. 5. – p. 93-101.

[12] Parker S. G. et al. RTSL: a ray tracing shading language //2007 IEEE Symposium on Interactive Ray Tracing. – IEEE, 2007. – p. 149-160.

[13] Perlin K. *An image synthesizer* //ACM Siggraph. – 1985. – . 19. – №. 3. – p. 287-296.

[14] Pharr M., Jakob W., Humphreys G. *Physically based rendering: From theory to implementation.* – Morgan Kaufmann, 2016.

[15] Sons K. et al. *shade.js: Adaptive Material Descriptions* //Computer Graphics Forum. – 2014. – Vol. 33. – №. 7. – p. 51-60.

[16] Stich M. *Real-time raytracing with Nvidia RTX*, GTC EU 2018

[17] Veach E. *Robust Monte Carlo methods for light transport simulation.* – PhD thesis : Stanford University, 1997. – . 1610.

[18] Zhdanov D. D., Ershov S.V., Deryabin N. B. *Object-oriented model of photo-realistic visualization of complex scenes* // Scientific visualization (in russian), Vol.5, № 4, 2013, pp. 88–117.

[19] Arnold renderer GPU demo press release. URL = https://www.arnoldrenderer.com/news/press-release-arnold-5-3-gpu/

[20] Blender community. *Cycles Open Source Production Rendering.* URL = https://www.cycles-renderer.org/

[21] Caustic Graphics. *OpenRL: Open Ray Tracing Language.* 2010.

[22] Houdini 17.5 VEX language reference. 2019. URL = https://www.sidefx.com/docs/houdini/vex/lang.html

[23] Hydra Renderer. Open source rendering system. KIAM RAS, MSU. 2019 URL = https://github.com/Ray-Tracing-Systems/HydraAPI

[24] Nvidia Turing arch. paper. 2019 URL = http://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[25] Nvidia RTX Ray tracing developer resources. 2019 URL = https://developer.nvidia.com/rtx/raytracing

[26] OctaneRender OSL Documentation. 2019. URL = https://docs.otoy.com/osl/

[27] Open Shading Language. 2019. URL = https://github.com/imageworks/OpenShadingLanguage

[28] Vulkan specification. 2019 URL = https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html