BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **14**, No 2

Sofia • 2014

Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2014-0018

A Framework for Visual Dynamic Analysis of Ray Tracing Algorithms

Hristo Lesev, Alexander Penev

Plovdiv University "Paisii Hilendarski", 4000 Plovdiv
Emails: hristo.lesev@gmail.com apenev@uni-plovdiv.bg

Abstract: *A novel approach is presented for recording high volume data about ray tracing rendering systems' runtime state and its subsequent dynamic analysis and interactive visualization in the algorithm computational domain. Our framework extracts light paths traced by the system and leverages on a powerful filtering subsystem, helping interactive visualization and exploration of the desired subset of recorded data. We introduce a versatile data logging format and acceleration structures for easy access and filtering. We have implemented a plugin based framework and a tool set that realize all ideas presented in this paper. The framework provides data logging API for instrumenting production-ready, multithreaded, distributed renderers. The framework visualization tool enables deeper understanding of the ray tracing algorithms for novices, as well as for experts.*

Keywords: *Visual, dynamic, analyzer, ray tracing, debugging.*

1. Introduction

Algorithms for ray tracing in computer graphics are widely used to generate production-ready images, animations and feature length films. Such rendering algorithms operate on huge amounts of data, and in recent years the need these algorithms to be physically correct has added additional difficulties with debugging and dynamic analysis of the algorithms in the ray tracing domain.

Our main goal is to create a modular architecture of a framework that enhances visualization and dynamic analysis of ray tracing algorithms in computer graphics domain.

Achieving this goal will require solving several tasks derived from the specific nature of the problem. A typical production-ready ray tracing system involves tracing billions of rays and takes a very long time to converge to a result. This involves an enormous amount of data flowing through the system and makes conventional debugging not a preferred option. Another aspect of the problem is that the complex recursive nature of the ray tracing algorithms hampers the step by step debugging of the system's workflow.

Like most computer graphics algorithms, ray tracing has a domain-specific concrete visual representation of the data. However, the existing symbolic debuggers do not reach the level of visual representation of the data in the computational domain of the algorithm that would be helpful to the developer.

Stochastic ray tracers rely heavily on random numbers and this makes static analysis rather useless. Ray tracing is parallel by nature and thus deprives the developers from the convenience of traditional debugging.

These challenges involve solving several different tasks, such as: recording the algorithm state at every stage of its execution, filtering huge volumes of data, and domain-specific data visualization. These tasks can be reflected directly into a set of subsystems that will build the desired framework.

Although our main goal is focused on helping developers in the implementation and analysis of computer graphics algorithms, we believe that the framework visual nature can be of great use in education.

We identify several target audiences that will benefit from using the described framework. These include:

- developers – for faster debugging of problematic algorithms and developing new algorithms;
- students – for better understanding of the ray tracing matter;
- end users – to help the support of the system by recording execution state logs.

2. Related work

Since the early days of the modern computer era, visual analysis has always been a field of intensive research. Modern integrated development environments (IDEs) have powerful tools for dynamic analysis of widely used algorithm fragments and data structures, but they are designed to work on a source code itself and thus cover a wide area of applications. In computer graphics domain the underlying data have a very specific visual representation, such as rays in 3D space, scene geometry, energy, time coherence, etc., that cannot be shown by general purpose debugging tools.

Most of the existing tools in the field of visualizing algorithms and data focus on the visualization of the abstract representations of the data structures implemented by the algorithm considered, such as trees, lists, arrays, graphs, etc. Another focus shift is towards program visualization and algorithm visualization. Such tools and systems are described in [12, 13, 14]. They indirectly show how the logic of the algorithm is working, but are not related in any way to the domain of

computation, in which the algorithm processes its data. This approach contributes little to the process of understanding complex algorithms in the computer graphics domain, such as ray tracing. The majority of the available visual frameworks are designed to simplify the creation of interactive tutorials and examples of how different algorithms operate with input data. However, very few frameworks are capable of doing dynamic analysis of these algorithms and thus they do not empower software developers. Rather than making snapshots of the runtime data on "interesting events" [18, 15], our approach leverages logging the algorithm's state and letting the end user dynamically filter only the subset he is interested in during the particular analysis, without the need to run the algorithm again.

Our intent is to reveal the processed data and the work done by the algorithm in the context of the problem it solves, by visualizing the closest physical equivalence of the virtual data. An example of such a system is given in [1], and this task is achieved by a highly visual approach of domain-specific data. Similar ideas are presented in [15] and [16].

One of the recent tools for domain-specific ray tracing data visualization is rtVTK [2]. Its concept is similar to our goals as it tries to implement unified programming interface for logging runtime data of ray-based algorithms execution, and builds a visual tool for analyzing the stored data. The main advantages of the toolkit are its plugin nature and layered visualizing, but on the other hand the filtering of the collected data is very vague. We think that filtering must have a central role in such a system, since it makes the visualization and analysis of huge data volumes possible. Modern production ray tracers are highly parallel and require adequate handling of the incoming log stream. Unfortunately, it is unclear whether rtVTK is capable of working in a parallel mode. The approach we choose for instrumentation of highly parallel algorithms, such as ray tracing, is similar to the ideas in [17].

In [3] and [4] powerful tools for realtime editing of the traced light paths based on visual interaction with the user, are given. They allow the selection of specific parts of the traced paths according to different filtering criteria, such as spatial position, ray type, pars of the scene interacted with the light, etc. These types of filtering are important to achieve our primary goal for better visualization of the data. The two papers throw no light on implementing such filters and in our opinion this is a must for this kind of systems. Being oriented mainly towards real time editing and focused on non physical correct modifications of the light transport, their work does not include logging and analysis of the runtime data.

In our implementation we use a modification of the light path notation described in [5] and [6] to filter specific light paths while doing visual analysis.

3. Implementation

The framework presented in this paper tries to enhance the process of development and analysis of computer graphics algorithms in three main areas: collecting data of the algorithm run time, easy searching and filtering through the stored information and visualizing the recorded data in the spatial domain of the computation.

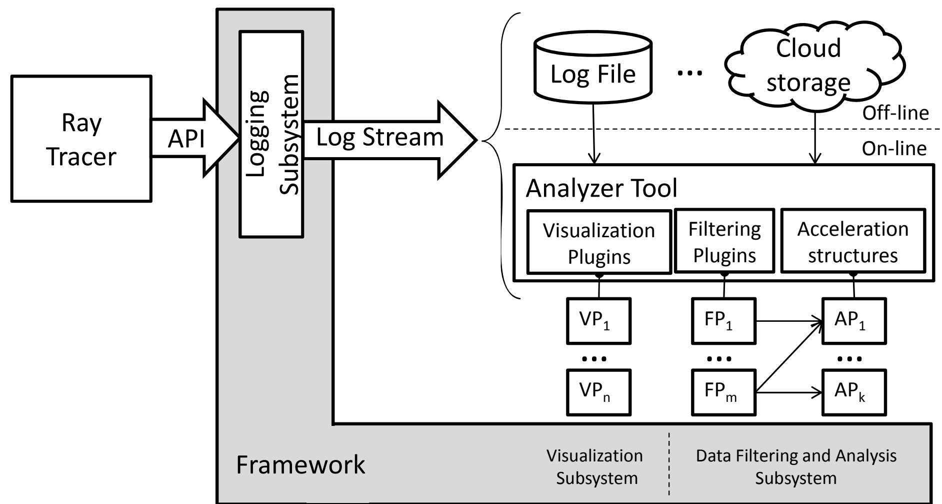


Fig. 1. System architecture

Each of these areas defines an autonomous subsystem in the framework architecture (Fig. 1). The tools currently implemented in the framework are focused on analyzing the stochastic ray tracing applications, but the plugin based architecture of the core enables the programmer to build new tools accurately, reflecting the analysis task that he/she faces. Each of the three subsystems provides its own extension interface.

3.1. Ray data logging subsystem

We introduce a simple data logging API for instrumentation existing rendering applications. The following pseudocode demonstrates the use of the API in a recursive ray tracer.

Pseudocode

```
for each pixel in the image
  pick a ray from the eye through the pixel
  trace(ray)

trace(ray)
  beginPacket(ray.ID)
  find nearest intersection with scene
  addToPacket(ray.ID, "parent", ray.parentID)
  addToPacket(ray.ID, "hitInfo", ray.hitInfo)
  shade(ray, ray.hitInfo.material)
  endPacket(ray.ID)

shade(ray, material)
  for each light source
    calculate direct light
    addToPacket(rayID, "light K contribution", light_energy)
    pick new random ray direction depending on material properties
    trace(newRay)
    addToPacket(rayID, "indirect contribution", indirect_energy)
```

A data logging subsystem uses three functions – **beginPacket** and **endPacket** for creating in-memory data structure for the packet, and **addToPacket** to store the arbitrary data that the developer might need to analyse later. The logging API has no additional requirements towards the instrumented ray tracer except for assigning a unique ID number for each newly spawned ray. Unique naming of the traced rays is common practice in today's renderers and we think this does not limit the developers using the framework in any way. Ray IDs are used to retrieve parent-child relations in the tree of the traced rays.

The current design of the data logging subsystem enables using the API in multithreaded and distributed renderers.

3.2. Data log streaming

Data captured during the logging phase is structured as a stream of data packets. The huge amounts of data due to the number of traced rays and the lack of consistency in packets' arriving time caused by the parallel execution environment lead to issues with scale and storage. In addition, we wanted our framework to be used for instrumenting production-ready distributed renderers, in which actual data storage might be on a different machine than the one running the ray tracer application. These constraints made the streaming data architecture a preferred solution.

Currently the framework implements three ways of handling a data log stream:

- storing the stream as a file for later analysis;
- transferring it in a cloud based storage;
- on-line – a real time view of the incoming packets in a visualization application.

Most of the time data logging is excessively used during the implementation and testing phase of the project lifecycle. Once the render system is deployed to a client's environment, the developer has little or no information about its operation and the problems encountered during its usage, thus making the remote support look like debugging a black box. We try to make this box a grey one by enabling the system to use a data logging API and send the recorded information to a secure cloud storage where it can be analysed later by the developer.

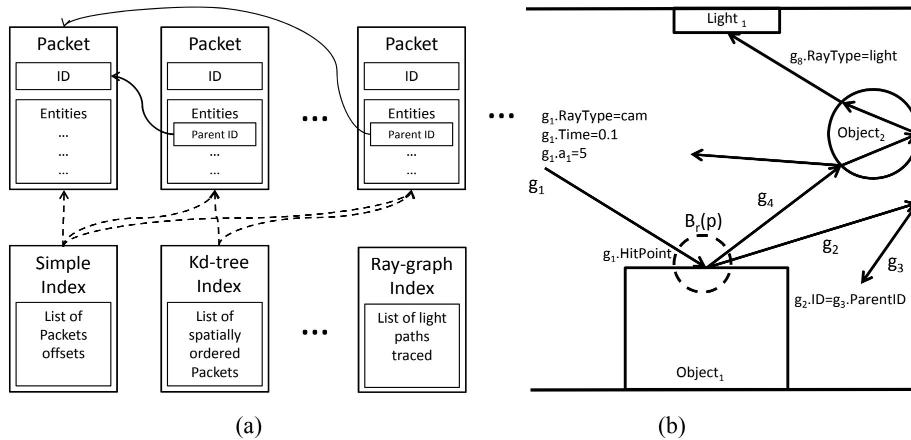
The data log stream consists of numerous packets, any of which having an arbitrary length. The packet is a key-value container that stores the state of the analysed algorithm. In the current implementation most packets contain information about rays traced through the scene, their position, direction, time, color, etc. Another usage of the packets is to store meta data about the scene, geometry, material properties, light sources, etc., and these packets are later used for visualizing the collected ray data within the spatial domain of computation.

Each packet consists of a packet header and a packet body (Fig. 2a). The header section contains the packet size and the unique ID in the data stream. The packet body contains the data payload represented as a key-value dictionary of the user defined attributes. When saved as a file, the packet body can be compressed to further minimize the disk space usage.

The framework does not impose any constraints on the data that can be stored as an attribute value. All relations among packages are also stored as attributes. For example, all packets that store traced rays are part of the light path graph and have to maintain a parent-child relation, where each child ray packets has an attribute with a value the ID of the parent packet (Fig. 2a).

When a complex scene is rendered, the number of stored packets can easily exceed a billion. This will make real time analysis and visualization of the stored data a difficult task to achieve. We address this issue by adding indices at the end of the stream. Building of the indices is done as a post processing operation after the data logging is complete. The indices are stored in the stream as packets.

The current implementation uses a simple packet start position index for easy finding of individual packets, kd-tree [8] – for spatial indexing when searching through the origins of the traced rays, and B-tree [9] – for searching through the rest of the packet attributes. A light path parent-child relation is kept in a separate graph based index, which allows easy extraction of all packets composing a valid subpath of the graph. The need for having these helper structures is determined by the expected high number of logged packets and the demand for filtering them as fast as possible, as well as the usage of the framework in real life scenarios (see the examples in 3.3).



(a) (b)

Fig. 2. Log stream structure (a); light paths filtering (b)

3.3. Ray packet filtering

Filtering can be done in several extensible ways. Here we shall use a simple predicate filtering by packet attributes and a combination of simple predicates.

Some of the most important filters we currently use are:

- the attribute values;
- the ray type;
- spatial filtration – regions of world space that rays hits;
- the time or timespan;
- ID of the thread that spawned this packet.

Next, we shall give more formal definitions of the filtering foundations implemented in the described system.

Let

$$G = (N, E);\ N = \{g_1, g_2, ..., g_n\},\ E \subseteq N \times N = \{\, e_i \in N \times N \mid \forall i = 1, ..., k \,\},$$

where G is a multigraph with a set of nodes N with elements g_i i=1, …, n, representing Data Log Packets. The edges in the graph (denoted as set E) are defined by ID-ParentID relations (and others) between the packets in the log.

For performing analysis in the ray tracing domain, one must consider the relations between all tracked rays forming ray paths in the traced graph.

Let

$$R_0 = (G, P_0) = \varnothing,$$

$$R_1 = (G, P_1) = \{(g_{i_1}) \in N \mid P_1(g_{i_1})\} \cup R_0,$$

$$R_2 = (G, P_2) = \{(g_{i_1}, g_{i_2}) \in N^2 \mid P_2(g_{i_1}, g_{i_2})\} \cup R_1, \ldots,$$

$$R_j = (G, P_j) = \{(g_{i_1}, g_{i_2}, ..., g_{i_j}) \in N^j \mid P_j(g_{i_1}, g_{i_2}, ..., g_{i_j})\} \cup R_{j-1}, \ldots,$$

where R is a relation between the nodes in G, set by the predicate P. These relations represent the part of the data that will be visualized and analysed.

Definition 1. An **Active Set** is a subset of all nodes in G.

Active Sets are used for visualizing data as a part of the Log analysis process. Active Sets are not required to be generated by any edges in the graph.

Definition 2. An **Active Path** is an ordered Active Set formed by nodes in G.

Active Path is a sequence of connected data packets. In the system described we use ID-ParentID attributes as relations between the nodes in the path. These relations represent rays generated sequentially during the ray tracing simulation of a light path.

Definition 3. A **Filter** is a predicate defining an Active Path/Active Set in graph G.

Relations R_j define a set of Active Paths with length less than or equal to j in graph G generated using a filter P_j.

Examples of different filters are given below, see also Fig. 2b:

(1) $$P_1 = (g_{i_1}) = (g_{i_1}.a_1 = 5) \wedge (g_{i_1}.a_2 > 3) \wedge (g_{i_1}.\text{Time} \in [0.1, 0.5])$$

In this example filter defining simple **Active Set** is shown using attributes a_1, a_2 and Time. In ray tracing domain the time of the ray is an important attribute for visualizing the process development in a certain time frame:

(2) $$P_2 = (g_{i_1}, g_{i_2}) = (g_{i_1}.a_1 > 5) \vee (g_{i_2}.a_2 = 6) \wedge (g_{i_1}.a_1 = g_{i_2}.a_1) \wedge$$
$$\wedge (g_{i_1}.\text{HitPoint} \in B_r(p)) \wedge (g_{i_1}.\text{ID} \in g_{i_1}\text{ParentID}).$$

Filter P_2 is defining **Active Path** by enforcing the requirement that the parent of g_{i_2} must be g_{i_1}. Another interesting part of the predicate is a spatial relation defined

by $B_r(p)$, it requires the starting point of the ray g_{i_1} to be in the volume enclosed by a sphere with radius *r* and center *p* (Fig. 2b):

(3) $$P_j = (g_{i_1}, g_{i_2}, ..., g_{i_j}) = (g_{i_1}.\text{RayType} = \text{cam}) \wedge (g_{i_2}.\text{RayType} = \text{ref}l) \wedge$$
$$\wedge (g_{i_3}.\text{RayType} = \text{ref}l) \wedge ... \wedge (g_{i_j}.\text{RayType} = \text{light}) \wedge$$
$$\wedge (g_{i_1}.\text{Ray} \cap \text{Object}_1 \neq \text{Ø}) \wedge (g_{i_2}.\text{Ray} \cap \text{Object}_2 \neq \text{Ø}) \wedge ... \wedge$$
$$\wedge (g_{i_1}.\text{ID} = g_{i_2}.\text{ParentID}) \wedge (g_{i_2}.\text{ID} = g_{i_3}.\text{ParentID}) \wedge ... \wedge$$
$$\wedge (g_{i_{j-1}}.\text{ID} = g_{i_j}.\text{ParentID}).$$

Filter P_j defines **Active Path** with length *j* and describes a common situation for filtering data gathered by ray tracing application. The first part of the predicate will extract a set of rays shot from the camera (**RayType=Cam**), reflected multiple times in the scene (**RayType=refl**) and finishing the path in a light source (**RayType=light**). There are also sub predicates that require specific rays like g_{i_1} and g_{i_2} to be bounced by concrete objects in the scene **Object$_1$** and **Object$_2$**. P_j is an equivalent to light path expressions described in [6].

The filter types are realized as plugins of the system (FP in Fig. 1). Some of them can use acceleration structures that can be stored as packets.

The filters can be configured using various user controls exposed by the Graphical User Interface (GUI). For example, we could use regular expressions, as shown in (3), to filter by the ray type or simple expressions for filtering packet's attributes, etc. All of the described configurations are presented to the user to easily build more sophisticated filtering predicates.

For faster searching, some filters can use different acceleration structures. The acceleration structures are generated once after the logging phase has finished and before the analysis has begun. They are added to the system as plugins (AP in Fig. 1). If one filter tries to use a certain acceleration structure, it asks the plugin system whether there is an acceleration plugin of this type and uses the result. Otherwise, the filter uses default searching logic. In general, one filter can use more than one acceleration structure to do different kinds of searches or to choose the optimal one depending on the current case. If the needed structure is not present in the log file, the filter plugin can enforce the acceleration plugin to create it and store it in the file.

The acceleration structures we have currently implemented in the system are:

- a simple index – for fast finding a packet by ID or other attributes. Used in simple filters generating **Active Sets** using filters like the one in (1);
- a point based kd-tree – for spatial searches. Used for fast constructing **Active Sets** using the spatial information stored in their elements, as shown in (2). It could be used in combination with other filters for extracting concrete **Active Paths**;
- a ray path graph – for extracting light paths from the ray trees, following the idea in example (3).

3.4. Logged data stream visualization

The visualization of the collected data in the domain of computation is of great importance when analyzing computer graphics algorithms. Plugin based architecture of the visualization subsystem enables the development of specific viewers (VP in Fig. 1) for different packet attributes the user wants to display.

We have implemented an interactive GUI application (Fig. 3a) for displaying the data stream. Rendering scene geometry and configuration is essential for visualizing traced rays in the context of the computation. Scene geometry, light sources and materials are obtained from the stream meta data packets. Currently, an OpenGL [10] rendering plugin is implemented for displaying the geometry on the screen. Procedural geometry types are supported via polygonization plugins communicating with the OpenGL viewer plugin.

The application is MVC based [11], which allows not only extending the data viewers and the available data representations but also implementing new ways for interacting with the scene, such as controller plugins.

The next step towards visual analysis is visualizing the traced Active Set and specific graph paths meeting specific filtering criteria defined by one or more filters (Definition 3). For every filtering plugin there is a corresponding application controller exposing the filter properties to the user. When the filtering configuration changes, the ray graph viewer is called to display the filtered Active Paths.

We have provided numerous built-in controllers for data filtering manipulation, including complex light path expressions as in (3), via GUI elements (Fig. 3b): sliders for controlling the visible timespan of the ray graph, 3D picking mode for selecting different parts of the geometry for filtering by ray-object interaction, sphere helper creation mode for defining 3D regions of the scene space for spatial filtering origins of the traced rays (2), a simple expression editor for attribute filtering, including ray types and ray subpaths (3).

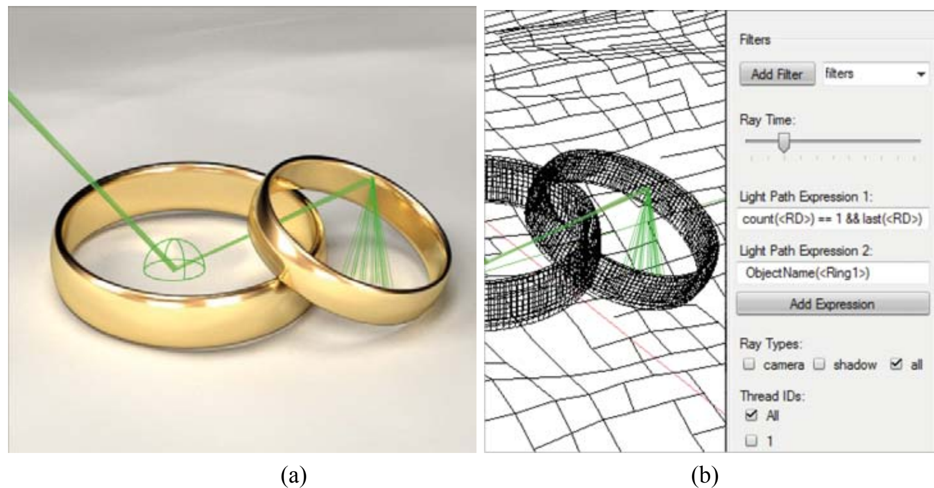
4. Results and applications

The described framework and its accompanying tools are implemented as a set of C# libraries that can be used by both managed and unmanaged applications. During the framework testing we have instrumented the experimental rendering system “RayTracer” [7] with the presented logging API. “RayTracer” is a multithreaded stochastic ray tracer with the support of physically correct materials composed of BRDFs, realistic light sources and the ability to combine different Global Illumination (GI) algorithms for calculating light propagation in the scene.

Tests were performed involving production heavy scenes and complex lighting setups (Fig. 3a). The overhead of the data logging measured in terms of render time is linear with respect to the number of the traced rays.

The framework main purpose is helping the developers who work in computer graphics domain to produce better quality rendering algorithms by visualizing the work done by the algorithm in the context of the computation.

Education is another area that can benefit from the described framework. Interactive visualization of the ray tracing algorithm was included as part of the computer graphics course for undergraduate students in Plovdiv University “Paisii Hilendarski”. We noticed that the ability to analyze recorded data and interact step by step with the implemented rendering algorithms provided the students with deeper understanding of how light-matter interaction is computed.



(a) (b)

Fig. 3. Screenshots of the visual analyzer tool: interactive light path visualization (a); filtering UI controls (b)

Interactive visualization allowed the students to animate the process of light propagation. This helped them to gain confidence to try and work on improving the rendering algorithms instead of just implementing the classical ones. The course instructors also benefit from the interactive visualization by having the chance to demonstrate the non intuitive parts of the rendering algorithms and to address the common pitfalls in their implementation.

The framework plugin architecture empowered us to implement different data filters and controls for best serving both developers and students (Fig. 3b). The option to send the collected data over the network proved invaluable for remote debugging and helped us to test the framework in a heterogeneous environment.

Of particular interest to us was the source code-logged state relation, in which the line number of the executed code is logged in every packet. This allows to highlight certain parts of the source to identify the lines responsible for some types of behaviour. This proved to be of great use for developers who are used to more classical source code debugging tools provided by the modern IDEs.

5. Conclusion

In this paper we proposed a plugin based software architecture for a framework for logging and dynamic analysis of computer graphics algorithms based on the ray tracing paradigm. The architecture proves to be very extensible and ready for

analyzing production heavy rendering systems. A powerful filtering subsystem helps the framework users to easily process huge volumes of collected data.

A fully functional prototype of the logging system was implemented based on the described architecture. The implementation allows the collection of data from multithreaded ray tracing algorithms executed on a single machine, as well as collection of data from distributed rendering environment over the network. The implementation strictly follows the plugin decomposition proposed in this paper.

The prototype constructively demonstrated the practical application of the proposed architecture. Several different filters were implemented for data extraction and analysis, including light path expression filtering, spatial filtering, time based filtering, executor thread ID filtering, etc.

We noticed that the interactive visualization tool of the framework and the implemented user controls for filter building were of great help for understanding how the analyzed algorithms work. This ability to interact with the data in the computational domain of the algorithm proves to be invaluable for developers in the debugging phase of the project and for students in the field of computer graphics algorithms.

In future numerous features can be introduced in the framework to further help its users. Such an extension is a differential filter which will work on two or more logs, analyzing their ray trees and showing how changing ray tracing parameters or the system source code affect the behaviour of the analyzed algorithm. Another valuable addition to the visualization subsystem would be an interactive ray tracer. It can easily visualize a different type of geometry consisting not only of triangles, but described by functions, such a ray tracer will eliminate the need for writing a visualization plugin for every new geometry type that we want to show during the analysis phase. We believe that implementing a simple ray tracer as a visualization plugin will help future developments.

Acknowledgements: The work is partially funded by the Fund Research of the Plovdiv University "Paisii Hilendarski" under Contract No NI13-FMI-002/2013.

References

1. Rusell, J. An Interactive Web-Based Ray Tracing Visualization Tool. Undergraduate Honors Program Senior Thesis. Department of Computer Science, University of Washington, 1999.
2. Gribble, C., J. Fisher, D. Eby, E. Quigley, G. Ludwig. Ray Tracing Visualization Toolkit. – In: Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ACM, New York, NY, USA, 2012, 71-78. doi: 10.1145/2159616.2159628.
3. Schmidt, T., et al. Path-Space Manipulation of Physically-Based Light Transport. – ACM Transactions On Graphics (TOG), Vol. **32**, 2013, No 4, Article 129, 129:1-129:8. doi: 10.1145/2461912.2461980.
4. Reiner, T., A. Kaplanyan, M. Reinhard, C. Dachsbacher. Selective Inspection and Interactive Visualization of Light Transport in Virtual Scenes. – Computer Graphics Forum / Eurographics, Vol. **31**, 2012, No 2, 711-718. doi: 10.1111/j.1467-8659.2012.03050.x.
5. Heckbert, P. S. Adaptive Radiosity Textures for Bidirectional Ray Tracing. – In: Proceedings of SIGGRAPH 1990, Vol. **24**, 1990, ACM, New York, USA, 145–154. doi: 10.1145/97879.97895.

6. OSL, visited 25.11.2013.
https://code.google.com/p/openshadinglanguage/wiki/LightPathExpressions
7. L e s e v, H. Photorealistic Computer Graphics In Informatics Education Process. – In: Proceedings of Conference "Education in the Information Society", Institute of Mathematics and Informatics of BAS and Plovdiv University "Paisii Hilendarski", Plovdiv, Bulgaria, 2010, 141-146.
8. S a m e t, H. The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1990.
9. B a y e r, R. The Universal B-Tree for Multidimensional Indexing: General Concepts. – In: Proceedings of the Worldwide Computing and Its Applications. Berlin, Heidelberg, Tsukuba, Japan, Springer, 1997, 198-209. doi: 10.1007/3-540-63343-X_48.
10. S h r e i n e r, D., G. S e l l e r s, J. M. K e s s e n i c h, B. M. L i c e a-K a n e. OpenGL Programming Guide: The Official Guide to Learning OpenGL. Version 4.3. 8th Edition. USA, Addison Wesley Professional, 2013.
11. K r a s n e r, G., S. P o p e. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. – Journal of Object-Oriented Programming, Vol. **1**, 1988, No 3, 26-49.
12. M y e r s, B. INCENSE: A System for Displaying Data Structures. – In: Proceedings of the ACM SIGGRAPH Computer Graphics, Vol. **17**, 1983, No 3, ACM, New York, NY, USA, 115-125. doi: 10.1145/800059.801140.
13. M u k h e r j e a, S., J. T. S t a s k o. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger. – ACM Transactions on Computer-Human Interaction (TOCHI), Vol. **1**, 1994, No 3, 215-244. doi: 10.1145/196699.196702.
14. C a r l i s l e, M., T. A. W i l s o n, J. W. H u m p h r i e s, S. M. H a d f i e l d. RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving. – ACM SIGCSE Bulletin, Vol. **37**, 2005, No 1, 176-180. doi: 10.1145/1047124.1047411.
15. N a p s, T. L., B. S w a n d e r. An Object-Oriented Approach to Algorithm Visualization – Easy, Extensible, and Dynamic. – ACM SIGCSE Bulletin, Vol. **26**, 1994, No 1, 46-50. doi: 10.1145/191033.191052.
16. N i k a n d e r, J., J. H e l m i n e n. Algorithm Visualization in Teaching Spatial Data Algorithms. – In: Proceedings of the Information Visualization, 11th International Conference, IEEE, Zurich, 2007, 505-510. doi: 10.1109/IV.2007.21.
17. K r a e m e r, E., J. T. S t a s k o. The Visualization of Parallel Systems: An Overview. – Journal of Parallel and Distributed Computing, Vol. **18**, 1993, No 2, 105-117. doi: 10.1006/jpdc.1993.1050.
18. B r o w n, M. Algorithm Animation. Cambridge, MA, USA, MIT Press, 1988.