The Gap between Visualization Research and Visualization Software (VisGap) (2021)
C. Gillmann, M. Krone, G. Reina, T. Wischgoll (Editors)

OSPRay Studio: Enabling Multi-Workflow Visualizations with OSPRay

Isha Sharma Dave DeMarle Alok Hota Bruce Cherniak Johannes Günther

Intel Corporation

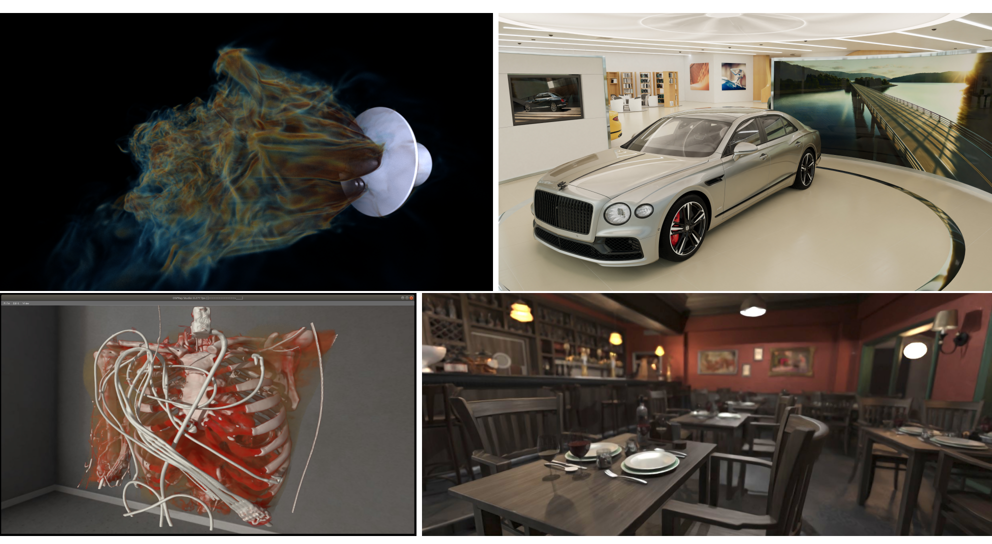

Figure 1: *Supporting workflows for Scientific, Product, Medical and Architecture Visualization with OSPRay Studio.*

Abstract

*There are a number of established production ready scientific visualization tools in the field today including ParaView [Aya15], VisIt [CBW*11] and EnSight [Ans]. However, often they come with well defined core feature sets, established visual appearance characteristics, and steep learning curves – especially for software developers. They have vast differences with other rendering applications such as Blender or Maya (known for their high-quality rendering and 3D content creation uses) in terms of design and features, and have over time become monolithic in nature with difficult to customize workflows [UFK*89]. As such a multi-purpose visualization solution for Scientific, Product, Architectural and Medical Visualization is hard to find. This is a gap we identify; and with this paper we present the idea of a minimal application called OSPRay Studio, with a flexible design to support high-quality physically-based rendering and scientific visualization workflows. We will describe the motivation, design philosophy, features, targeted use-cases and real-world applications along with future opportunities for this application.*

CCS Concepts

• ***Computing methodologies*** → ***Rendering;*** • ***Software and its engineering*** → ***Designing software;***

1. Motivation

Besides and within the field of Scientific Visualization there are a number of specific application domains including Medical Visualization, Architectural Visualization, Engineering Visualization, Product Visualization, and Geographic and Climate System Visualization to name a few, see Figure 1 for examples. A number of systems have been built for both general purpose and domain specific users.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

DOI: 10.2312/visgap.20211086

Each domain has different requirements and expectations for the visual quality of the images produced by these tools. In Medical and Engineering Visualization the clarity of interpretation is the norm, with non-physical color mapping and simple lighting. In Architectural and Product Visualization, photorealism with physically based lighting and materials are required. Often the practitioners of these fields become committed to specific workflows and as a result, software tools tend to get enriched mostly with features complementing that workflow. With long-term institutional development the tools grow and become harder to modify, to accommodate new and different workflows and new visual aspects.

In all cases, a deluge of data means that large scale parallel data processing is helpful; and in some cases, it is essential for effective visualization. At the same time high quality rendering is a computationally expensive proposition that benefits greatly from fine-grained multi- and many-core parallelism.

OSPRay [WJA*17] is an intermediate level ray tracing library with built-in support to take advantage of MPI, thread and SIMD level parallelism to attain interactive rendering rates on large scale data sets. It offers "scivis" quality renderings and path traced photorealism. OSPRay is Open Source, has a modular and lightweight implementation, and is evolving quickly.

A handful of established general purpose visualization tools including ParaView [Aya15] and VisIt [CBW*11] have recently been extended to incorporate OSPRay. Unfortunately, with their large code bases, expansive feature sets, and complex development, the feature set of OSPRay that is exposed in these tools does not keep pace with that of OSPRay itself.

Hence, the idea of the OSPRay Studio project originated. There was a strong need for a lightweight end-user application with a set of commonly used scivis features, that enables different types of visualization workflows. The design for OSPRay Studio is meant to support each workflow with a common underlying design. It further aims to support many new proof-of-concept workflows and investigating single solutions that scale from desktop uses to data centers.

2. Design

OSPRay Studio mainly consists of two components: an *application* for defining user-interaction and a *library* for implementing its internal scene state. The implementation of both these components utilizes the design goals mentioned below. Its design philosophy is a single implementation supporting multiple workflows and it has originated from its motivation. Following are the four main design goals of OSPRay Studio:

1. *Lightweight*
   The main implementation should consist of only commonly used visualization features, enabling multiple workflows with a single design.
2. *Functionality Abstraction*
   The main object structure for internal scene layout should be independent of the functionality. By doing so, different functions can be applied to the same object structure. For example, different rendering backends can be used to generate visualizations of the same scene or different rendering outputs can be generated from the same scene.
3. *Extensibility*
   The core implementation should be extensible with advanced features, which contributes to a cleaner overall design by keeping the main implementation to a minimum.
4. *Scalability*
   Rendering capabilities of the application should be scalable, i.e., should enable rendering on end-user workstations or laptops as well as distributed rendering across HPC clusters. For this goal OSPRay studio currently relies on its rendering backend: OSPRay.

In the following we discuss some of the design constructs and concepts implemented in OSPRay Studio together with the goal they aim to accomplish.

2.1. Abstract Scene Graph

An *Abstract Scene Graph* (for simplicity we will just use *scene graph* from now on) represents the internal scene structure of OSPRay Studio. It is implemented using a Directed Acyclic Graph (DAG).

A scene graph consists of scene objects in a DAG representation, where every object is represented as a *node* and has at least one parent (unless it is root). Each node (except for leaf nodes) can have any number of children. For example, a light object can be represented as a light node in the scene graph, having a transform node as parent to define its position in the world.

The scene graph can be rendered using a particular rendering implementation. Its current and only rendering implementation is using the OSPRay backend. A rendering implementation is responsible for converting the scene graph to a representation understood by the rendering backend. Thus, the scene graph is a different scene structure than its renderer scene hierarchy and allows for loose coupling between the two. This allows for customization of current scene objects like lights, camera, ..., and introduction of new objects in the scene during rendering time. With every change, the backend scene hierarchy is updated and new frames are received from OSPRay.

A scene graph makes for a versatile scene representation that is open for binding with different rendering implementations for multiple backends and rendering time customizations. It can also be used to save the current state of the scene. A saved scene graph can act as a *portable visualization state* that contains all scene objects like lights, cameras, etc. In OSPRay Studio we can save the current scene graph as an `.sg` file, which contains the scene objects in JSON-format. With editable `.sg` files a scene graph can even be modified offline. This not only allows one to save an otherwise hard to achieve visualization state, but also keeps state of a concrete scene similar across multiple instances of the application. A small excerpt from a saved `.sg` file is shown in Figure 2. It lists only a single scene object, i.e., a transform, as child of an importer object.

Because scene graph allows separating the scene hierarchy from actual rendering implementation, it allows us to fulfill the second design goal of *functionality abstraction*.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

```
{
 "children": [
  {
   "description": "<no description>",
   "name": "imperial_crown_rootXfm",
   "subType": "Transform",
   "type": 9,
   "value": {
    "affine": [ 0.0, 0.0, 0.0 ],
    "linear": {
     "x": [ 1.0, 0.0, 0.0 ],
     "y": [ 0.0, 1.0, 0.0 ],
     "z": [ 0.0, 0.0, 1.0 ]
    }
   }
  }
 ],
 "description": "<no description>",
 "filename": "AustrianCrown/impCrown.obj",
 "name": "impCrown.obj.obj_importer",
 "subType": "importer_obj",
 "type": 20
}
```

Figure 2: *Excerpt of an* `.sg` *file, which captures the state of the scene graph in JSON-format.*

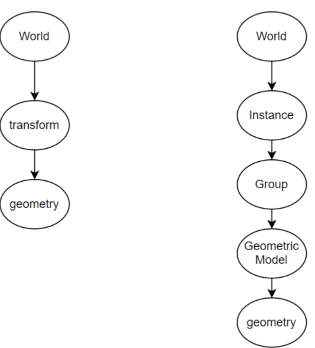

Figure 3: *Difference in scene hierarchy for adding a simple geometry to the world between OSPRay Studio (left) and OSPRay (right). In the abstract scene graph representation of OSPRay Studio we have fewer objects.*

2.2. Scene Graph Library

The *Scene Graph (SG)* library consists of different types of *node* classes. These classes are used to instantiate nodes for creating a scene graph. Following are some of the main node classes:

generic This is the base node class from which all other node classes derive. It provides common node functions such as `createChild` for creating a child of current node or `add` for adding some other node to the current node. New node classes can be derived from the generic node class and enhanced with special features. For example, the importer base class derives from it and has further special derived importer classes for specific file types.

strongly-typed These node types implement specific data types, e.g., `string` or `int`, that hold the corresponding data values. These nodes are commonly used to specify properties and hold values for them for other nodes. For example, a `float` node for specifying the radius of a sphere can be created and added to a spheres geometry node, specifying a common property all spheres.

OSPRay-typed These classes create OSPRay objects internally and store a handle to them. For example, the perspective camera node creates an OSPRay perspective camera and sets its value to the handle of the OSPRay camera object. There is no 1:1 correspondence of the scene graph to the OSPRay scene hierarchy. Hence not all OSPRay scene object types will have a corresponding SG node class. The idea is to keep the resulting scene graph loosely coupled with the renderer and to hide the complexity of OSPRay's scene hierarchy. Therefore, objects that are not part of the scene graph but that are needed by OSPRay for rendering, e.g., GeometricModel or Instance, are created only when the rendering function is invoked, see Figure 3.

The scene graph is a design concept and the SG library provides its implementation via the node classes and their API. Thus, it fulfills the design goal of *functionality abstraction*.

2.3. Visitor Paradigm

Visitors are classes that implement functions to be performed on the scene graph, once it is created. Hence the visitor API also contributes to the *functionality abstraction* design goal by keeping those functions separate from scene graph.

Visitors form part of the design pattern that allows different operations to be performed on different elements in a hierarchical object structure. In OSPRay Studio, visitor classes are implemented to perform node-specific tasks on the scene graph during traversal, which happens in a post-order fashion. They also keep the node classes relatively lightweight by abstracting functionality from them. An example implementation of a visitor would be the renderer-specific scene hierarchy generation. OSPRay Studio currently has one such visitor called `RenderScene` for creating the complete OSPRay scene hierarchy. As a second example, the `GenerateImguiWidgets` visitor creates widgets with property editors for scene graph nodes during traversal.

2.4. Lightweight GUI and Widgets

Most visualization use-cases will require a fast graphical user interface (GUI) to modify data or scene properties. To follow the design goal of keeping the application implementation *lightweight*, OSPRay Studio uses Dear ImGui [oco] for creating its GUI and menus on top of a GLFW window. Dear ImGui is a lightweight and fast graphical user interface library for C++. It is used for creating both the main menu and widgets in OSPRay Studio. The main menu consists of File, Edit and View menu options, each with its own set of functions like node editing, scene graph viewers, etc.

The main GUI comes with commonly used features, e.g., to allow users to import data and clear data, change scene node properties via editors, or save frame output in different formats. It is also easy to minimize it, to avoid cluttering the view of the scene, and users can use keyboard shortcuts for several functions. Overall, the implementation of the GUI is lightweight and intended to be augmentable or replaceable. It also comes with test scenes to showcase different features.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

Widgets provide custom GUI controls for modifying node properties or for specific actions like animating a scene. They extend the main GUI and provide feature specific controls, e.g., Transfer Function widgets can be used alongside Volumetric data to change the transfer function property and yet do not have to be a part of the main GUI.

In Figure 4 the main window of the application with some of the GUI features is shown. The drop-down of the *Edit* menu option presents different editors to change scene node properties. One of them is the *Lights* editor, where we can alter properties of individual lights that are in the scene. An *Animation* widget is loaded by default only for animated scenes, allowing us to extend the GUI functionality without cluttering the main GUI. This demonstrates the *extensibility* aspect of our design goals and keeps common functionality separated from more advanced features.

Figure 4: *OSPRay Studio main GUI with animation widget. The vividly animated Buster Drone scene was created by LaVADraGoN [LaV16].*

2.5. Operation Modes

OSPRay Studio was designed with multiple *modes* of operation, as a strategy to enable different use-cases and workflows. Presently there are three modes:

GUI provides per default an application window with menu
batch allows for quickly writing a single image or sequence of output images from the command-line terminal
timeseries for rendering temporal data

Each mode enables vital use-cases, which are further discussed in Section 3. Modes also make proof-of-concept workflows plausible and hence also contribute to the *extensibility* design goal. For example, a future mode could be a *Headless* mode, which provides rendering as a service functionality by serving rendering requests over the network.

2.6. Plugins

Plugins are used for extending both OSPRay's and OSPRay Studio's functionality. For example, rarely used importers with deep library dependencies can be implemented as a plugin rather than adding them to the core application, to keep the main application size small. Plugins also allow experts to add custom functionality to OSPRay Studio and they are thus a key component for the *extensibility* design goal. OSPRay Studio comes with an *Example Plugin* to show how plugins work. However, no plugin is built or enabled by default, but should be selected manually. Plugins are built as shared libraries which can be loaded dynamically at runtime.

2.7. Distributed Rendering with MPI

OSPRay Studio relies on OSPRay's distributed frame buffer for cluster scalable distributed rendering [UWA*19]. There are currently two modes of scalable rendering. In offload mode, the aggregated compute power of the cluster is used to accelerate rendering via sort-first technique of relatively small data sets. In distributed mode the aggregated memory capacity of the cluster brings greater data scalability via sort-last rendering technique, with the caveat that photorealism is not supported (because efficient handling of incoherent secondary rays very challenging with distributed data). By enabling such distributed rendering with MPI OSPRay Studio meets the *scalability* design goal.

3. Case-Studies

OSPRay Studio's flexible design and lightweight code base enabled visualization demos at conferences like ACM SIGGRAPH and other use-cases by collaborators. Presented below are some of such case-studies that reflect on the philosophy of supporting multiple-workflows and might have been challenging on production visualization tools.

3.1. NASA FUN3D-based Dataset

Using a new plugin for implementing an importer for NASA's FUN3D [NAS] fluid solver output format, we were able to import large fluid simulation volumes unaltered into OSPRay Studio for visualization as a timeseries.

NASA's supersonic retropropulsion simulation for Mars [EKD*14] is one of those FUN3D-based datasets. This dataset simulates the jet flow of a theoretical Mars lander when braking during entry in Martian atmosphere, see Figure 5. The simulation data contains 1000 total timesteps, where each timestep is a large unstructured grid containing 143.4 million vertices, which define 788.8 million cells and 1.1 million surface polygons forming the lander. Each vertex contains seven simulation variables.

The FUN3D importer reads from the simulation data into memory as scene graph volume objects without modifying or preprocessing the data. The volumes (and other surfaces) are then rendered with OSPRay interactively with frame rates between 5–20 fps, depending on transfer function and view. We tested the visualization on an 8-node cluster; each node contained 2 × Intel Xeon 8280L CPUs, 384 GB DRAM and 3 TB PMEM.

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

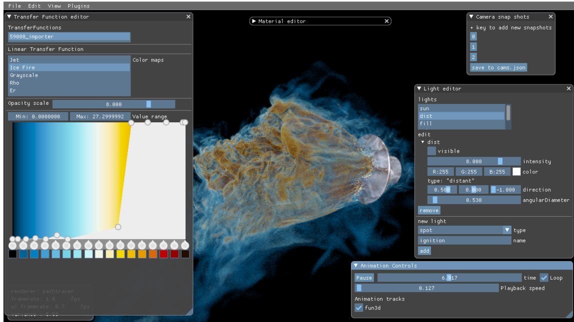

Figure 5: *NASA retropropulsion dataset as rendered using FUN3D plugin of OSPRay Studio.*

3.2. Cloudscape SIGGRAPH 2020 Demo

OSPRay Studio was used as the application front-end for showcasing the scalable multi-server rendering of a cloudscape during SIGGRAPH 2020. For this demo the *timeseries* mode with multiple framebuffers and the *GUI* mode with MPI distributed rendering of OSPRay were used. Two types of clouds were simulated in SideFX Houdini with VDB volumes: fractus and mediocris clouds. Clouds have light scattering properties and variable density, making them a challenging proposition to render and hence the choice for this demo.

With OSPRay Studio's *timeseries* mode we were able to showcase interactive individual timeseries evolution of 16 different clouds used in the final cloudscape and quickly manipulate the look of each timestep, see Figure 6. With main GUI editors, we were able to change any renderer or scene settings for the look of the demo. The series of 16 individual clouds consumed ½ TB of memory. To keep all the data readily available, Intel Optane persistent memory was used.

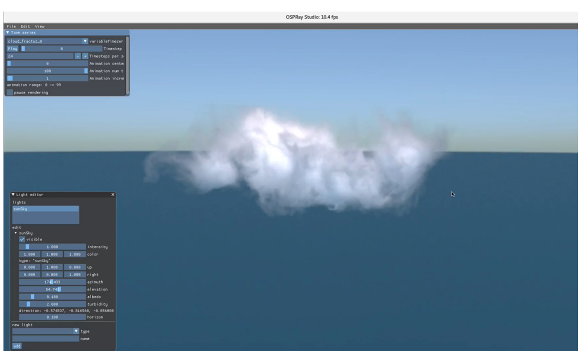

Figure 6: *Timeseries progression of a single fractus cloud from SIGGRAPH 2020 Demo.*

A single large 3D scene in VDB format combining all the clouds and their instances was created to demonstrate the cloudscape with distributed rendering, see Figure 7. The final scene was roughly 30 GB large and was rendered with high-quality renderer settings. We used a 10-node cluster where each node contained 2 × Intel Xeon 8280L CPUs, 384 GB DRAM and 3 TB PMEM to achieve interactive frame rates.

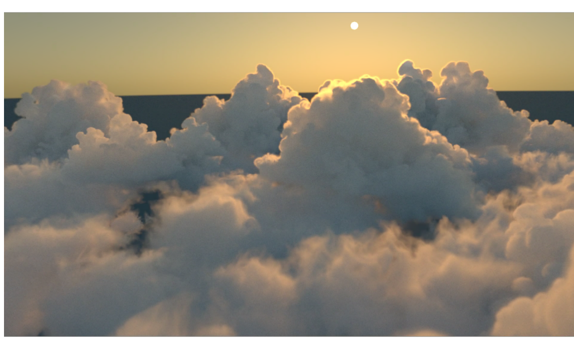

Figure 7: *The Cloudscape demo of combined VDB cloud volumes at SIGGRAPH 2020.*

3.3. Autonomous Driving Project

OSPRay Studio will be potentially used in the tool-chain for synthetic data-generation for an Autonomous Driving Safety and Assurance project in collaboration with the German automobile industry and funded partially by the German Government [KI 20]. The goal of the project is AI validation for Autonomous Driving systems. A physically-based synthetic data generation pipeline provides many advantages for this purpose [GGSB19]. Some features of OSPRay Studio that are used by the toolchain include:

glTF importer for loading large outdoor scenes with architectural assets, vehicles, vegetation, pedestrians etc.

Photo-realistic rendering of scenes with physically-based materials and global illumination. Animation and skinning of pedestrians (Figure 8), vehicles etc.

Figure 8: *glTF model of a pedestrian with animation and skinning as rendered in OSPRay Studio.*

Image sequences of rendered scene animation generated with Batch Mode are used as training or validation data for AI models.

Supporting simulation of optical sensors like LIDAR. With LIDAR light support from OSPRay, we can render any glTF asset as if it was observed by a LIDAR light source and render as point cloud in OSPRay Studio. Currently, this generated point cloud data

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

can be exported as a `.pcd` file in Batch mode and can be simultaneously viewed using the PCD importer which implements support for viewing PCD files, see Figure 9.

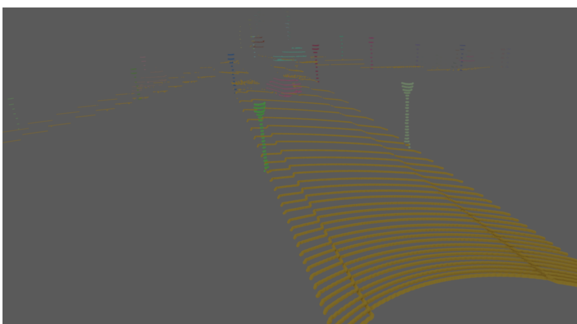

Figure 9: *LIDAR output of a glTF scene as viewed in OSPRay Studio with PCD importer.*

Supporting generation of Ground Truth data with a *Metadata* plugin which exports metadata information of a glTF scene. By exporting such information in EXR images as additional layers, we can effectively generate Ground Truth images for validation, see Figure 10, which shows a sample EXR image with metadata using tev (The EXR Viewer [Tom]). The following types of data can be currently exported:

- in EXR: id (instanceID), objectID, depth, worldPosition
- 3D bounding boxes as JSON export

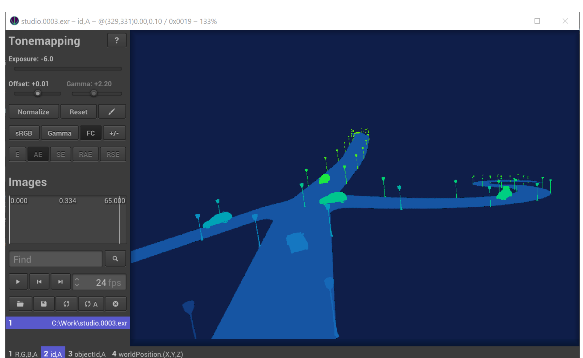

Figure 10: *An EXR image with additional layers to store metadata of Ground Truth data.*

3.4. Bentley Motors Collaboration

As a proof-of-concept for Bentley Motors, OSPRay Studio was used for a car configurator which enables users to interact with a realistic car model and select from different customizable options for their car, see Figure 11. This requires high-fidelity rendering of the car models with physically-accurate materials and lighting. The car models are complex and created with authentic 3D data used to manufacture the vehicle, with no data processing like triangle decimation etc. Each car model consists of more than 25 million triangles. User interaction with these complex car models is enabled at interactive frame rates with changes to the car taking effect smoothly.

Figure 11: *PoC for using OSPRay Studio in car configuration of Bentley Vehicles.*

Bentley also used OSPRay Studio for their virtual showroom configuration, with a total of 12 Bentley cars and other high complexity props making the showroom scene well over 330 million triangles, see Figure 12.

Figure 12: *Bentley virtual showroom as rendered in OSPRay Studio with OSPRay.*

3.5. Medical Visualization

OSPRay Studio has been used to demonstrate path traced volume rendering of medical datasets such as the one shown in Figure 13. In Medical Imaging, a growing use case is to provide realistic looking volumes or surfaces representing bone, muscle or skin with customizable lighting support. When combined with camera animation this can assist in communication with patients, training materials and other presentations. OSPRay's and thus OSPRay Studio's support for implicit isosurface rendering within path traced volume rendering was used to represent bone surfaces in the following renderings.

3.6. Architectural Visualization

The BISTRO scene from Amazon Lumberyard, Open Research Content Archive (ORCA) [Lum17], is an example of an architectural dataset with photorealistic materials such as cement, glass etc., illumination from multiple sources such as environment, street

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

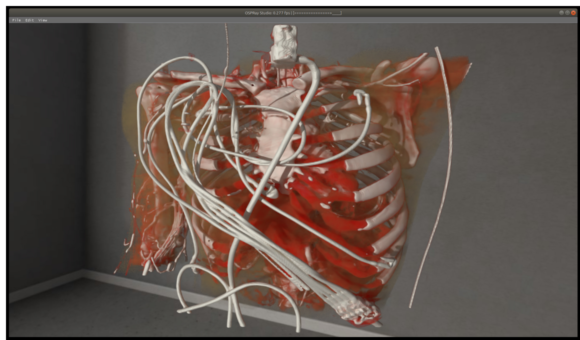

Figure 13: *Path traced Volume rendering of Cardiac CT dataset [CTA20] with iso-surfaces.*

bulbs, . . . and complex geometry such as vegetation, bistro cutlery, ... It is 4.1 million triangles for both the interior and exterior models combined and was rendered in OSPRay Studio, see Figure 14. This scene demonstrates scalability of interactive photorealistic rendering within OSPRay Studio.

Figure 14: *Scalable photorealistic rendering of Amazon Lumberyard BISTRO scene indoor (top) and outdoor (bottom).*

3.7. Engineering Visualization

As a proof-of-concept for an Engineering Visualization use case, OSPRay Studio application was used for volumetric rendering of an example silicon layout structure. In Figure 15 we show the example structures as rendered in OSPRay Studio using OSPRay and Open VKL [Inta]. Open VKL is a collection of high-performance volume computation kernels, for performance-optimized volume traversal and sampling functionality for a variety of volumetric data.

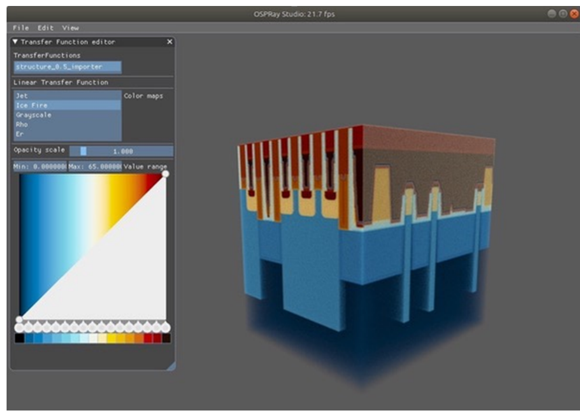

Figure 15: *Visualization of silicon layout structure through OSPRay Studio using OSPRay and OpenVKL volume rendering.*

3.8. Large Scale Stellar Outburst Simulation

OSPRay Studio was used as the application for rendering of large-scale volumetric stellar radiation simulation data from Argonne National Laboratory and UC Santa Barbara in a SIGGRAPH 2019 Demo. It is a 1.5 TB volumetric data set which was path traced using OSPRay and Intel Open VKL features with Intel Select Solution clusters and Intel Optane DC Persistent Memory in OSPRay Studio, see Figure 16.

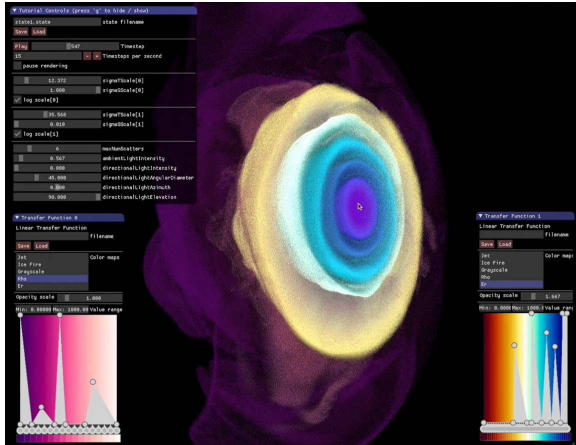

Figure 16: *SIGGRAPH 2019 Stellar Radiation Visualization: Volumetric Path Tracing with OSPRay Studio, OSPRay v2.0 and Open VKL.*

4. Conclusion and Future Directions

OSPRay Studio has been instrumental in many PoC workflows as described by various case studies above and it continues to develop while keeping the design goals in consideration. We have thus found that such a minimal core implementation, with supporting architecture for extensibility, can support a multitude of visualization use-cases by adhering to loose coupling between functionality, object structure and rendering backend. It also makes OSPRay

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

Studio an open and flexible platform for testing new ideas and features with greater ease as compared to other big monolithic solutions. The core implementation, being small, also facilitates easier learning for a new user and much easier adoption. Hence, it will continue to push for closing the gap between special purpose rendering applications.

Given its inception is recent, there is scope for adding many features. Some potential next features and future direction for development include:

Rendering implementation for other backends. For example, implementing an ANARI [Khr] backend for OSPRay Studio. This could become an alternative to its current OSPRay backend and provide flexibility for choosing between renderer backends.

Application level distributed rendering for providing scalable solutions for renderers that do not have distributed rendering capabilities.

Quantitative data analysis for visualization. Additional widgets to support heatmaps, data transformations, and zone picking would enhance OSPRay Studio as a data analysis tool rather than just a data presentation tool. Most of its current data operations happen in the rendering phase, hence we are actively working on more post-processing options like image operations.

Establishing a service infrastructure around OSPRay Studio's PoC Headless Mode. This will enable visualization over the web and multi-client collaboration for example.

Other improvements include: support for most commonly used 3D/4D data formats, simple movie creations with keyframing, scivis rendering with materials, support for in-situ visualizations, bindings for enabling scene graph creation from other language interfaces, e.g., python bindings etc. Some of these features are in progress.

OSPRay Studio developers are actively working on some of the future-directions discussed above and as mentioned before it is an Open Source project and we invite collaboration from all interested parties via OSPRay Studio GitHub [Intb].

Acknowledgment

Nikolay Zhavoronok and Daniil Fadeev from Intel for sharing silicon structure sample models. Sean McDuffee from Intel for creating cloud assets in SideFX Houdini

References

[Ans] Ansys EnSight. URL: https://www.ansys.com/products/fluids/ansys-ensight. 1

[Aya15] AYACHIT U.: *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., Clifton Park, NY, USA, 2015. 1, 2

[CBW*11] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., BONNELL K., MILLER M., WEBER G., HARRISON C., PUGMIRE D., FOGAL T., GARTH C., SANDERSON A., BETHEL E. W., DURANT M., CAMP D., FAVRE J., RÜBEL O., NAVRATIL P., VIVODTZEV F.: VisIt: An end-user tool for visualizing and analyzing very large data. *Proceed SciDAC* (01 2011), 1–16. 1, 2

[CTA20] CTA-cardio.nrrd from slicer testing data mirror, May 2020. URL: https://github.com/Slicer/SlicerTestingData/releases. 7

[EKD*14] EDQUIST K. T., KORZUN A. M., DYAKONOV A. A., STUDAK J. W., KIPP D. M., DUPZYK I. C.: Development of supersonic retropropulsion for future mars entry, descent, and landing systems. *Journal of Spacecraft and Rockets 51*, 3 (2014), 650–663. doi:10.2514/1.A32715. 4

[GGSB19] GÜNTHER J., GRAU O., SHARMA I., BRUECHER B.: Advantages of physically based rendering for autonomous driving validation. In *Proceedings of the 3. ACM Computer Science in Cars Symposium (CSCS)* (October 2019). 5

[Inta] Intel Open Volume Kernel Library. URL: https://www.openvkl.org. 7

[Intb] Intel OSPRay Studio. URL: https://github.com/ospray/ospray_studio. 8

[Khr] Khronos ANARI. URL: https://www.khronos.org/anari. 8

[KI 20] KI Absicherung – Safe AI for Automated Driving, 2020. URL: https://www.ki-absicherung-projekt.de/en/. 5

[LaV16] LAVADRAGON: Buster drone, September 2016. URL: https://sketchfab.com/3d-models/buster-drone-294e79652f494130ad2ab00a13fdbafd. 4

[Lum17] LUMBERYARD A.: Amazon lumberyard bistro, open research content archive (ORCA), July 2017. URL: http://developer.nvidia.com/orca/amazon-lumberyard-bistro. 6

[NAS] FUN3D. URL: https://fun3d.larc.nasa.gov/. 4

[oco] OCORNUT: Dear ImGui. URL: https://github.com/ocornut/imgui. 3

[Tom] TOM94: tev – The EXR Viewer. URL: https://github.com/Tom94/tev. 6

[UFK*89] UPSON C., FAULHABER T. A., KAMINS D., LAIDLAW D., SCHLEGEL D., VROOM J., GURWITZ R., VAN DAM A.: The application visualization system: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications 9*, 4 (1989), 30–42. doi:10.1109/38.31462. 1

[UWA*19] USHER W., WALD I., AMSTUTZ J., GÜNTHER J., BROWNLEE C., PASCUCCI V.: Scalable ray tracing using the distributed framebuffer. *Computer Graphics Forum 38*, 3 (2019), 455–466. URL: https://www.ospray.org, doi:https://doi.org/10.1111/cgf.13702. 4

[WJA*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRATIL P.: OSPRay – A CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 931–940. doi:10.1109/TVCG.2016.2599041. 2

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.