Intel® oneAPI Rendering Toolkit

OSPRay Studio

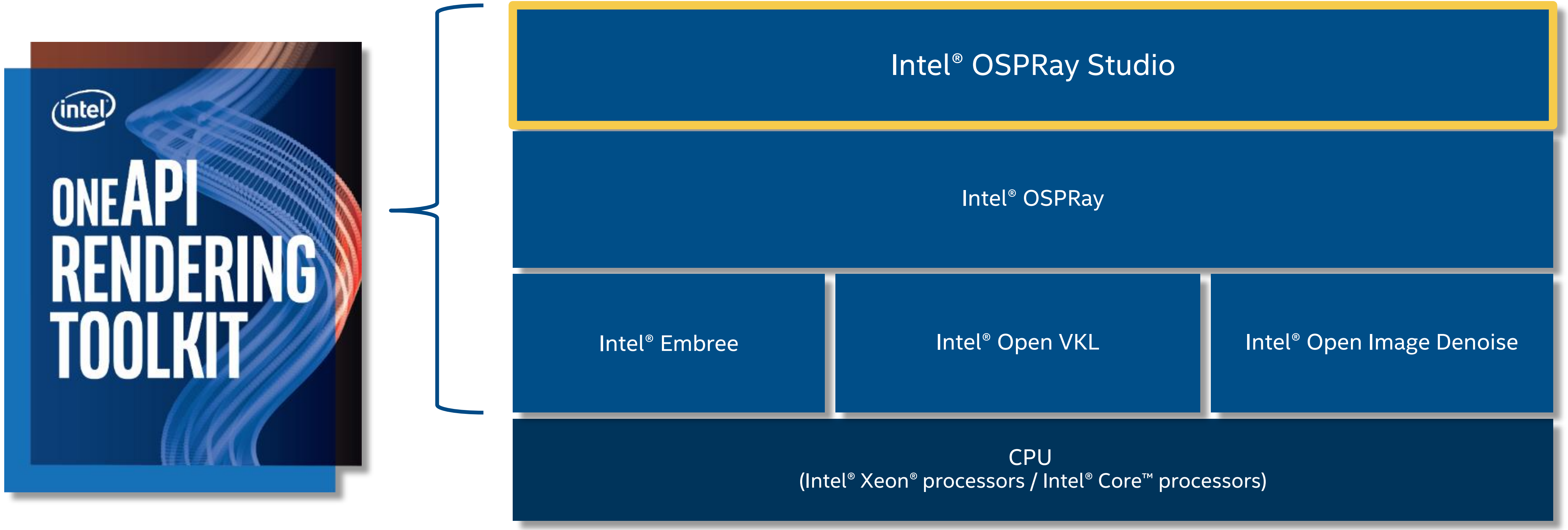
Isha Sharma

Agenda

- Intel® OSPRay Studio overview
- Feature Highlights
- *Interactive Demo*: Introduction
- OSPRay Studio Design
- *Interactive Demo*: Animation
- Python Bindings; concept and Demo
- *Interactive Demo*: Scalable Rendering with MPI

Intel® OSPRay Studio overview

A simple-to-use scene graph-based application for driving all of OSPRay's features



Intel® OSPRay Studio

Intel® OSPRay

Intel® Embree

Intel® Open VKL

Intel® Open Image Denoise

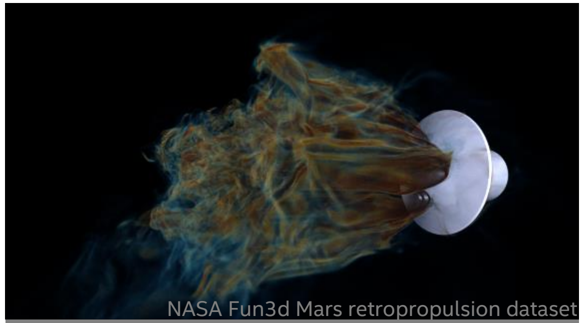
CPU
(Intel® Xeon® processors / Intel® Core™ processors)

Intel® OSPRay Studio

Feature Highlights

- **File Importers** – obj/mtl, glTF, vdb, structured and unstructured volume formats
- **Scene and image exporter** – saves rendered frames in different image formats. Or save the scene file with materials/lights properties of objects.
- **Scene Graph Library** – a library of node classes and visitors classes to create and render a scene graph, including Animation and Skinning.
- **Plugins** – runtime loadable shared-object libraries that can extend many aspects of the scene graph and application UI
- **Modes** – different ways of interaction with scene. Example, GUI or Batch mode of interaction.
- **GUI and Widgets** – standard GUI and custom GUI controls which extend the main GUI and provide feature specific controls

Visualization in the broader term..

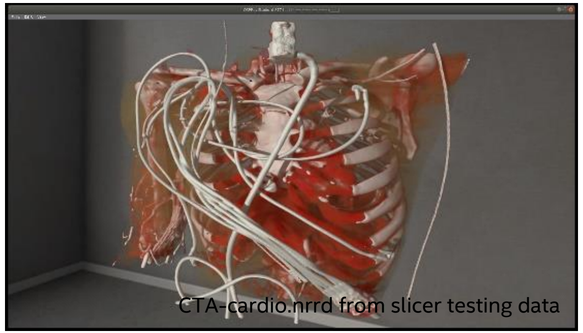


NASA Fun3d Mars retropropulsion dataset

Scientific Visualization

Bentley Motors Ltd. Vehicle Models used with permission

Product Visualization



CTA-cardio.nrrd from slicer testing data

Medical Visualization

Amazon lumberyard bistro

Architecture Visualization

Intel® OSPRay Studio

Building OSPRay Studio

- Make sure you have *OSPRay Superbuild*. For more information: https://github.com/ospray/ospray#cmake-superbuild
- Export following variables to install locations of the superbuild:
  - *ospray_DIR , openvkl_DIR, embree_DIR, rkcommon_DIR*
- Clone OSPRay Studio
  - git clone https://github.com/ospray/ospray_studio/
- Create build directory and change directory to it (we recommend keeping a separate build directory)
  - *cd ospray_studio && mkdir build && cd build*
- Then run the typical CMake routine
  - cmake .. && make -j `nproc`
  - Set up LD_LIBRARY_PATH (on Linux) or DYLD_LIBRARY_PATH (on macOS) correctly to contain all dependencies

Intel® OSPRay Studio

Interactive Demo: Introduction

OSPRay Studio Design

Components

- **Application** – defining user-interaction for eg: GUI application called MainWindow
- **Scene Graph Library** – a *library* for implementing its internal scene representation

Scene Graph Library

- The Scene Graph(SG) library implements the Abstract Scene Graph(or simply a Scene Graph) representation of the scene. It contains:
  - *Nodes* classes, used to instantiate nodes for creating a Scene Graph.
  - *Visitors* classes, to perform node specific tasks to be performed on the scene graph, once it is created.

Scene Graph Library

- Types of Node classes :
  - **Generic**: Base node class, implements important functions like createChild() and add().
  - **Strongly-typed**: Implements specific data-types like string or int.
  - **OSPRay-typed**: creates OSPRay objects internally and stores a handle to them.
- Nodes are connected in a **parent-child** tree structure,
- The root Frame node contains **Framebuffer**, **Camera**, **Renderer**, and **World** children – which are the main objects of the OSPRay ospRenderFrame() API.
- More documentation here: https://github.com/ospray/ospray_studio/blob/master/doc/scenegraph.md

Abstract Scene Graph

- An *Abstract Scene Graph* is a design concept and represents the internal scene structure of OSPRay Studio.
- Directed Acyclic Graph(DAG) of scene objects
- Every object is represented as a *node* and has at least one parent (unless it is root)

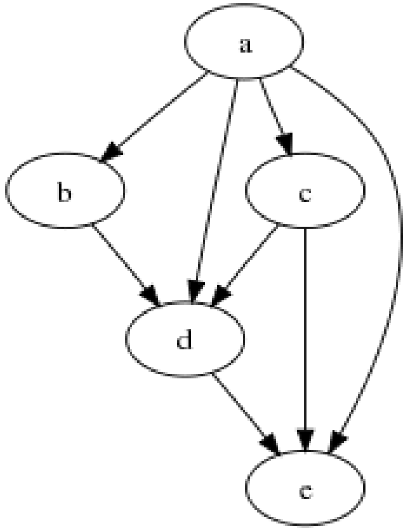


Fig: A Directed Acyclic Graph

Abstract Scene Graph

- 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.

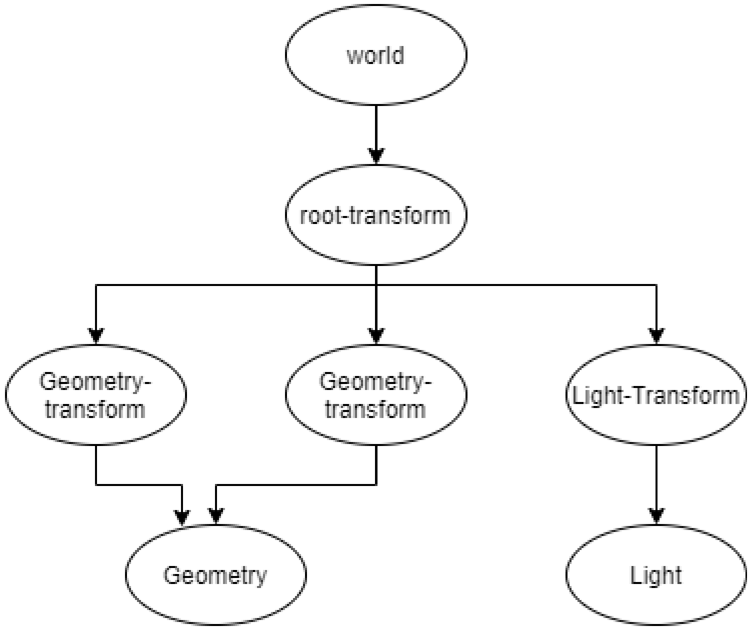


Fig: DAG implementation of scene graph

Abstract Scene Graph

- Different scene structure than its renderer scene hierarchy
- Scene graph can be rendered using a particular rendering implementation(Visitors)
- Allows for loose coupling between the two scene representations
- Customization of scene objects like lights, camera
- Introduction of new objects in the scene during rendering time
- Backend scene hierarchy is updated simultaneously, and new frames are received from OSPRay.

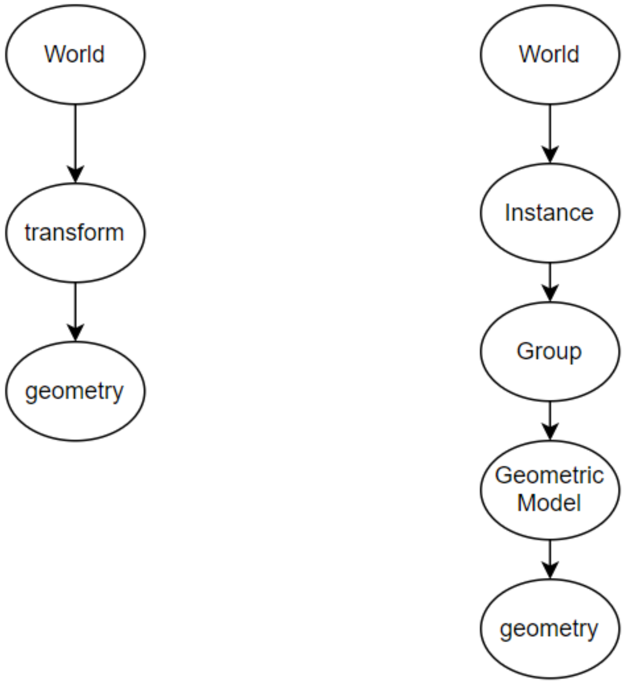
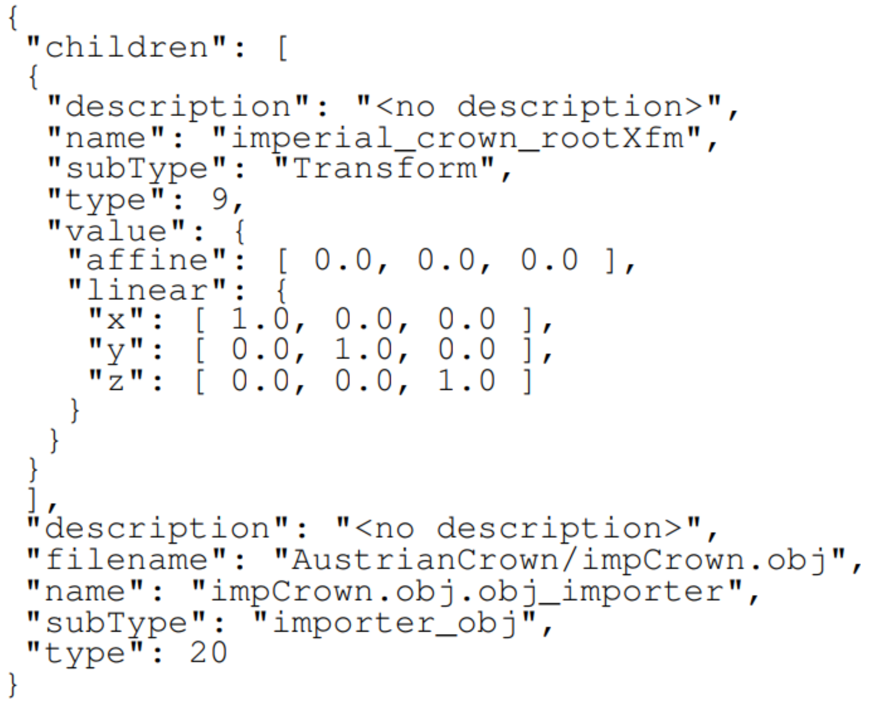


Fig: Difference in scene hierarchy for adding a simple geometry to the world between OSPRay Studio (left) and OSPRay (right). In the abstract scene graph representation of OSPRay Studio we have fewer objects

Saving a scene graph

- *portable visualization state* that contains all scene objects like lights, cameras, etc.
- scene objects in JSON-format.
- .sg files are editable i.e.; modify a scene graph offline



```
{
 "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
}
```

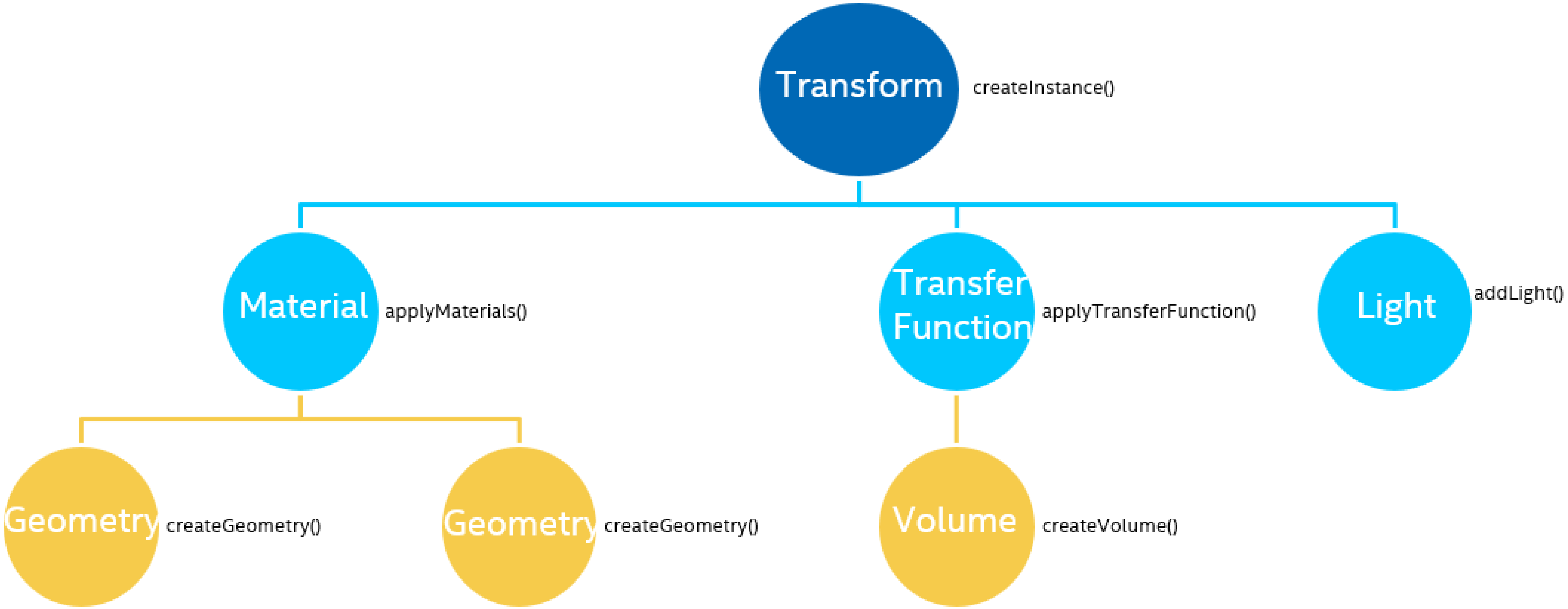
Fig: excerpt from a saved .sg file

Visitors

Design pattern to implement different operations on different elements in a hierarchical object structure.

- Classes for implementing the rendering backends. Only OSPRay implemented currently.
- Converts scene graph to a representation understood by the rendering backend.
- **Commits** scene graph data into a format that the OSPRay API expects
- Can also perform other node specific tasks like generate Widgets for every node

Visitors example: RenderScene()



Intel® OSPRay Studio Plugins

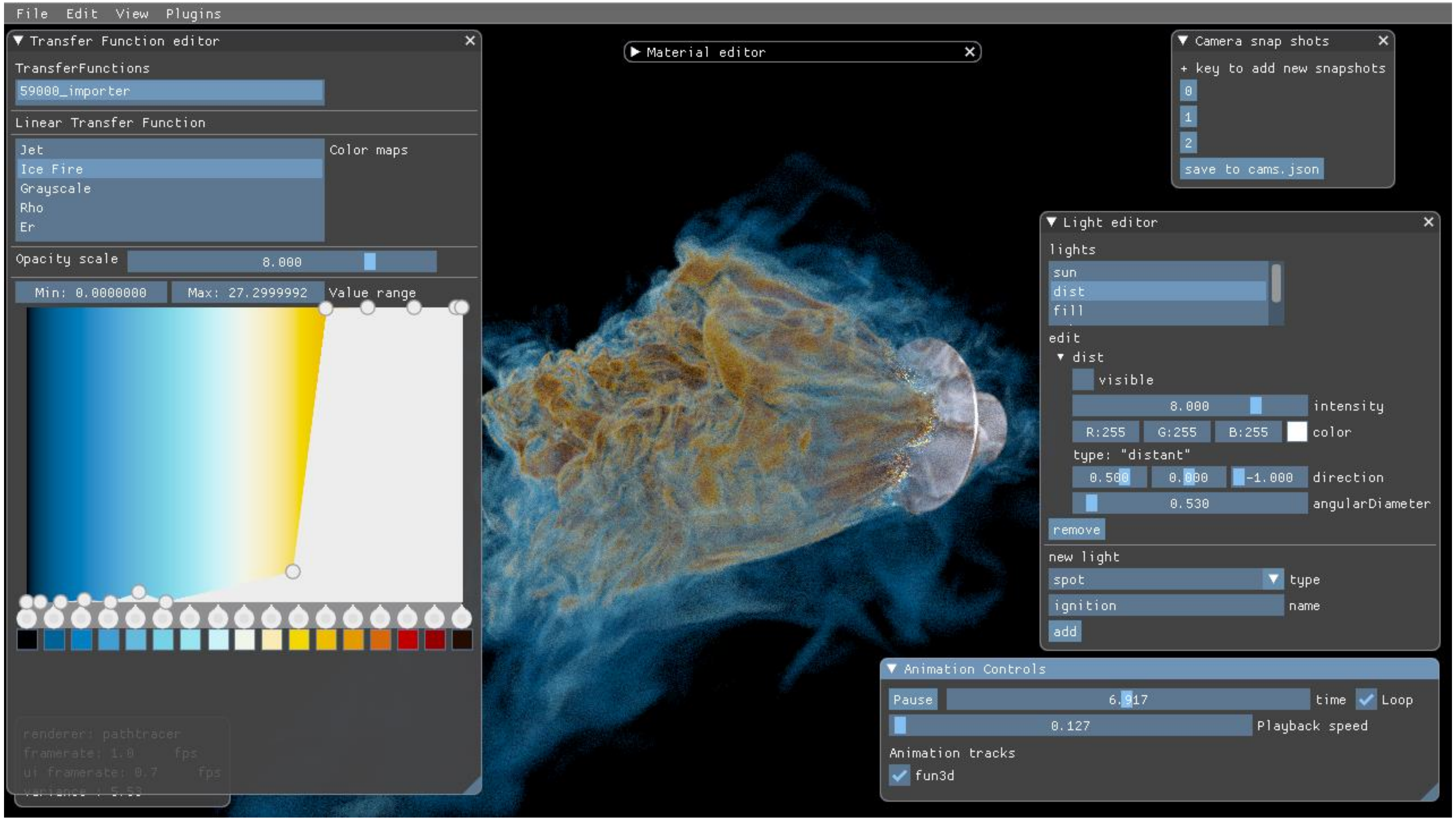
Customizations through plugins:

Shared-object libraries loaded at runtime

- Used for implementing new features, example: file loaders, UI menus and panels, scene graph nodes (ex. add new geometry, volumes, materials).
- Inherit from the **Plugin** class and implement **init_plugin_<name>()** function
- The derived Plugin class can implement either a UI panel or a mainMethod()
- The UI elements on these panels allow the plugin to interact with the user and scene graph.
- Plugins can also register new scene graph nodes to create new scene graph functionality.

Scientific Visualization: NASA Fun3D data

NASA Fun3d Mars retropropulsion dataset https://fun3d.larc.nasa.gov/



▼ Transfer Function editor

TransferFunctions

59000_importer

Linear Transfer Function

Color maps: Jet, Ice Fire, Grayscale, Rho, Er

Opacity scale 8.000

Min: 0.0000000   Max: 27.2999992   Value range

renderer: pathtracer
framerate: 1.0 fps
ui framerate: 0.7 fps
variance : 5.53

▼ Camera snap shots

+ key to add new snapshots

0
1
2

▼ Light editor

lights: sun, dist, fill

edit

▼ dist

□ visible

8.000 intensity

R:255 G:255 B:255 color

type: "distant"

0.500 0.000 -1.000 direction

0.530 angularDiameter

new light

spot type

ignition name

▼ Animation Controls

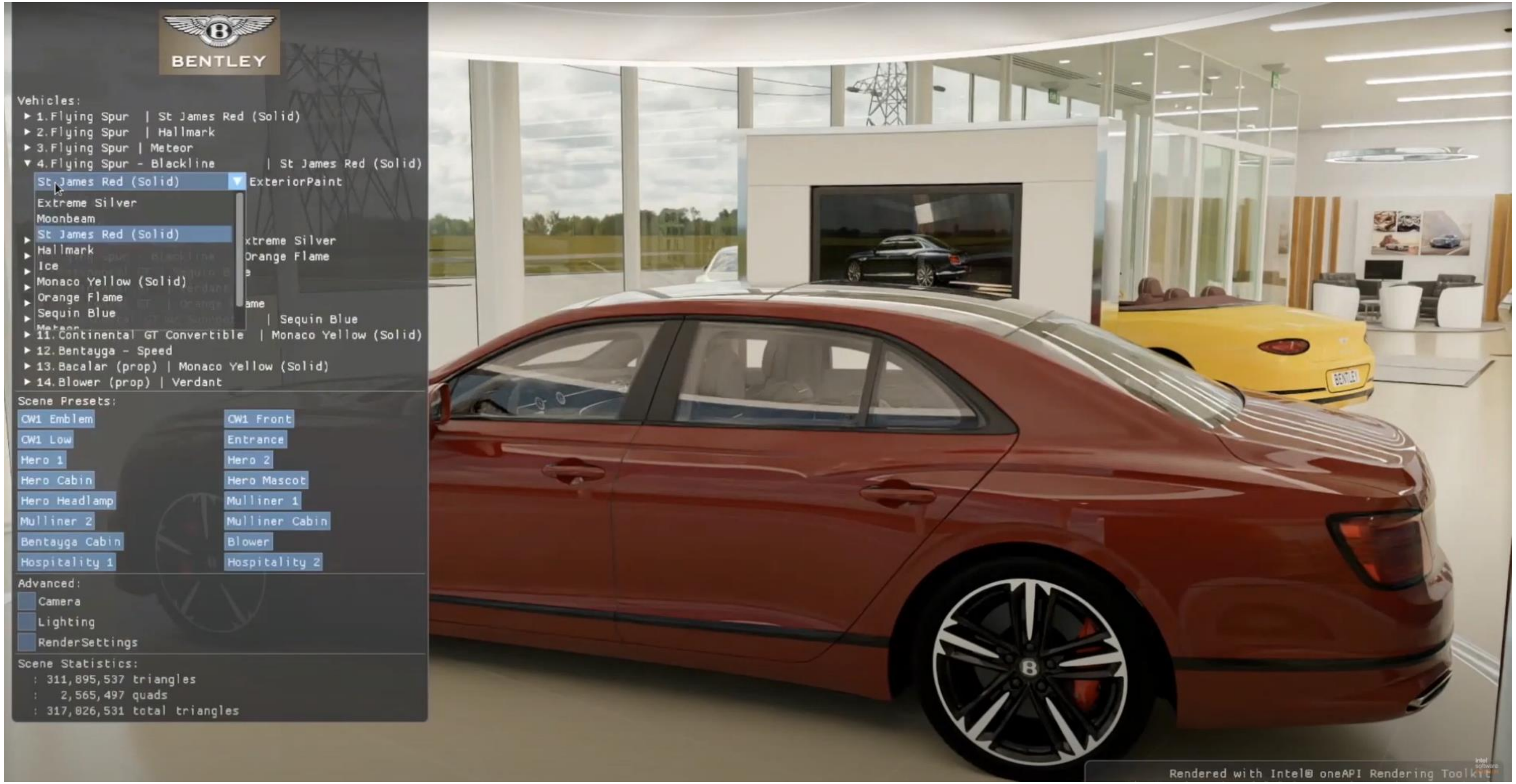
6.917 time ☑ Loop

0.127 Playback speed

Animation tracks

☑ fun3d

Product Visualization: Bentley Motors collab

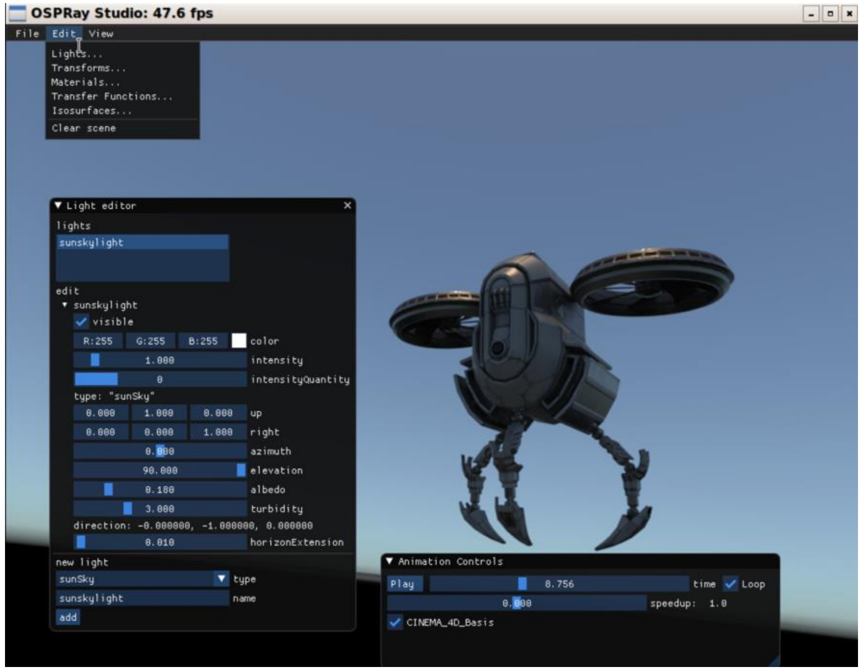


```
Scene Statistics:
  : 311,895,537 triangles
  :   2,565,497 quads
  : 317,026,531 total triangles
```

Bentley Motors Ltd. Vehicle Models used with permission

GUI and Widgets

- Fast GUI to modify scene properties
- Dear ImGui for GUI and menus atop a GLFW window
- Used for creating both main menu and widgets
- Easy to minimize , avoids cluttering
- Widgets provide custom GUI controls like animation controls



lights
sunskylight

edit
▼ sunskylight
☑ visible
R:255 G:255 B:255 color
1.000 intensity
0 intensityQuantity
type: "sunSky"
0.000 1.000 0.000 up
0.000 0.000 1.000 right
0.000 azimuth
90.000 elevation
0.180 albedo
3.000 turbidity
direction: -0.000000, -1.000000, 0.000000
0.010 horizonExtension

new light
sunSky type
sunskylight name

8.756 time ☑ Loop
0.000 speedup: 1.0
☑ CINEMA_4D_Basis

Intel® OSPRay Studio

Interactive Demo: Animation

Python Bindings

- Preliminary implementation of python bindings of the OSPRay Studio scene graph library using pyBind11.
- Creates a Python module that can be imported in your python code and used directly.
- Allows you to call functions and pass data from Python to C++

Python Bindings

Short excerpt from the implementation:

- Include pybind11 and sg library headers
- Use PYBIND11_MODULE macro to create the *py::module* object
- *sg.def()* defines a function that’s exported by the bindings, meaning it will be visible from Python.

```
PYBIND11_MODULE(pysg, sg)
{
  // OSPRay initialization /////////////////

  sg.def("init", &init);

  // Main Node factory function////////////


  sg.def(
     "createNode", py::overload_cast<std::string,
std::string>(&createNode));
  sg.def("createNode",
     py::overload_cast<std::string, std::string,
rkcommon::utility::Any>(
         &createNode));
....
}
```

Python Bindings

- Calling the exported functions from python:
- Import *pysg*, name should be same as used with PYBIND11_MODULE macro.
- Call functions/classes exported directly, here we call function *createChild()*

```
#sgTutorial.py

import pysg as sg
from pysg import Any, vec3f, Data, vec2i

sg.init(sys.argv)

W = 1024
H = 768

window_size = Any(vec2i(W, H))

frame = sg.Frame()
frame.createChild("windowSize", "vec2i", window_size)
world = frame.child("world")
..
..
```

Intel® OSPRay Studio

Interactive Demo: Scalable Rendering with MPI