OSPRAY

AN OPEN, SCALABLE, PARALLEL, RAY TRACING BASED RENDERING ENGINE FOR HIGH-FIDELITY VISUALIZATION

Version 1.4.3
January 10, 2018

Contents

1 OSPRay Overview **4**
- 1.1 OSPRay Support and Contact . . . 4
- 1.2 Version History . . . 4

2 Building OSPRay from Source **14**
- 2.1 Prerequisites . . . 14
- 2.2 Compiling OSPRay on Linux and Mac OS X . . . 15
- 2.3 Compiling OSPRay on Windows . . . 16

3 OSPRay API **17**
- 3.1 Initialization . . . 17
  - 3.1.1 Command Line Arguments . . . 17
  - 3.1.2 Manual Device Instantiation . . . 17
  - 3.1.3 Environment Variables . . . 19
  - 3.1.4 Error Handling and Status Messages . . . 19
  - 3.1.5 Loading OSPRay Extensions at Runtime . . . 20
- 3.2 Objects . . . 20
  - 3.2.1 Parameters . . . 21
  - 3.2.2 Data . . . 21
- 3.3 Volumes . . . 23
  - 3.3.1 Structured Volume . . . 23
  - 3.3.2 Adaptive Mesh Refinement (AMR) Volume . . . 24
  - 3.3.3 Unstructured Tetrahedral Volumes . . . 24
  - 3.3.4 Transfer Function . . . 25
- 3.4 Geometries . . . 26
  - 3.4.1 Triangle Mesh . . . 26
  - 3.4.2 Spheres . . . 26
  - 3.4.3 Cylinders . . . 27
  - 3.4.4 Streamlines . . . 27
  - 3.4.5 Isosurfaces . . . 28
  - 3.4.6 Slices . . . 28
  - 3.4.7 Instances . . . 28
- 3.5 Renderer . . . 28
  - 3.5.1 SciVis Renderer . . . 29
  - 3.5.2 Path Tracer . . . 30
  - 3.5.3 Model . . . 30
  - 3.5.4 Lights . . . 30
  - 3.5.5 Materials . . . 33
  - 3.5.6 Texture . . . 38
  - 3.5.7 Texture Transformations . . . 38
  - 3.5.8 Cameras . . . 39
  - 3.5.9 Picking . . . 43
- 3.6 Framebuffer . . . 43
  - Pixel Operation . . . 45

3.7 Rendering 45

4 Parallel Rendering with MPI **46**
4.1 Prerequisites for MPI Mode 46
4.2 Enabling the MPI Module in your Build 46
4.3 Modes of Using OSPRay's MPI Features 46
4.4 Running an Application with the "offload" Device 47
4.5 Running an Application with the "distributed" Device 48

5 Examples **49**
5.1 Tutorial 49
5.2 Example Viewer 50
5.3 Distributed Viewer 50
5.4 Demos 51

Chapter 1

OSPRay Overview

OSPRay is an **o**pen source, **s**calable, and **p**ortable **ray** tracing engine for high-performance, high-fidelity visualization on Intel® Architecture CPUs. OSPRay is released under the permissive Apache 2.0 license.

The purpose of OSPRay is to provide an open, powerful, and easy-to-use rendering library that allows one to easily build applications that use ray tracing based rendering for interactive applications (including both surface- and volume-based visualizations). OSPRay is completely CPU-based, and runs on anything from laptops, to workstations, to compute nodes in HPC systems.

OSPRay internally builds on top of Embree and ISPC (Intel® SPMD Program Compiler), and fully utilizes modern instruction sets like Intel® SSE4, AVX, AVX2, and AVX-512 to achieve high rendering performance, thus a CPU with support for at least SSE4.1 is required to run OSPRay.

1.1 OSPRay Support and Contact

OSPRay is under active development, and though we do our best to guarantee stable release versions a certain number of bugs, as-yet-missing features, inconsistencies, or any other issues are still possible. Should you find any such issues please report them immediately via OSPRay's GitHub Issue Tracker (or, if you should happen to have a fix for it,you can also send us a pull request); for missing features please contact us via email at `ospray@googlegroups.com`.

For recent news, updates, and announcements, please see our complete news/updates page.

Join our mailing list to receive release announcements and major news regarding OSPRay.

1.2 Version History

1.2.1 Changes in v1.4.3:

- Several bug fixes
  - Fixed potential issue with static initialization order
  - Correct compiler flags for Debug config
  - Spheres `postIntersect` shading is now 64-bit safer

1.2.2 Changes in v1.4.2:

- Several cleanups and bug fixes
  - Fixed memory leak where the Embree BVH was never released when an OSPModel was released

  - Fixed a crash when API logging was enabled in certain situations
  - Fixed a crash in MPI mode when creating lights without a renderer
  - Fixed an issue with camera lens samples not initilized when spp <= 0
  - Fixed an issue in ospExampleViewer when specifying multiple data files

- The C99 tutorial is now indicated as the default; the C++ wrappers do not change the semantics of the API (memory management) so the C99 version should be considered first when learning the API

1.2.3 Changes in v1.4.1:

- Several cleanups and bug fixes
  - Improved precision of ray intersection with streamlines, spheres, and cylinder geometries
  - Fixed address overflow in framebuffer, in practice image size is now limited only by available memory
  - Fixed several deadlocks and race conditions
  - Fix shadow computation in SciVis renderer, objects behind light sources do not occlude anymore
  - No more image jittering with MPI rendering when no accumulation buffer is used
- Improved path tracer materials
  - Also support RGB `eta`/`k` for Metal
  - Added Alloy material, a "metal" with textured color
- Minimum required Embree version is now v2.15

1.2.4 Changes in v1.4.0:

- New adaptive mesh refinement (AMR) and unstructured tetrahedral volume types
- Dynamic load balancing is now implemented for the `mpi_offload` device
- Many improvements and fixes to the available path tracer materials
  - Specular lobe of OBJMaterial uses Blinn-Phong for more realistic highlights
  - Metal accepts spectral samples of complex refraction index
  - ThinGlass behaves consistent to Glass and can texture attenuation color
- Added Russian roulette termination to path tracer
- SciVis OBJMaterial accepts texture coordinate transformations
- Applications can now access depth information in MPI distributed uses of OSPRay (both `mpi_offload` and `mpi_distributed` devices)
- Many robustness fixes for both the `mpi_offload` and `mpi_distributed` devices through improvements to the `mpi_common` and `mpi_maml` infrastructure libraries
- Major sample app cleanups
  - `ospray_sg` library is the new basis for building apps, which is a scenegraph implementation
  - Old (unused) libraries have been removed: miniSG, commandline, importer, loaders, and scripting
  - Some removed functionality (such as scripting) may be reintroduced in the new infrastructure later, though most features have remained and have been improved

  - Optional improved texture loading has been transitioned from ImageMagick to OpenImageIO

- Many cleanups, bug fixes, and improvements to `ospray_common` and other support libraries
- This will be the last release in which we support MSVC12 (Visual Studio 2013). Future releases will require VS2015 or newer

1.2.5 Changes in v1.3.1:

- Improved robustness of OSPRay CMake `find_package` config
  - Fixed bugs related to CMake configuration when using the OSPRay SDK from an install
- Fixed issue with Embree library when installing with `OSPRAY_INSTALL_DEPENDENCIES` enabled

1.2.6 Changes in v1.3.0:

- New MPI distributed device to support MPI distributed applications using OSPRay collectively for "in-situ" rendering (currently in "alpha")
  - Enabled via new `mpi_distributed` device type
  - Currently only supports `raycast` renderer, other renderers will be supported in the future
  - All API calls are expected to be exactly replicated (object instances and parameters) except scene data (geometries and volumes)
  - The original MPI device is now called the `mpi_offload` device to differentiate between the two implementations
- Support of Intel® AVX-512 for next generation Intel® Xeon® processor (codename Skylake), thus new minimum ISPC version is 1.9.1
- Thread affinity of OSPRay's tasking system can now be controlled via either device parameter `setAffinity`, or commandline parameter `osp:setaffinity`, or environment variable `OSPRAY_SET_AFFINITY`
- Changed behavior of the background color in the SciVis renderer: `bgColor` now includes alpha and is always blended (no `backgroundEnabled` anymore). To disable the background don't set bgColor, or set it to transparent black (0, 0, 0, 0)
- Geometries "`spheres`" and "`cylinders`" now support texture coordinates
- The GLUT- and Qt-based demo viewer applications have been replaced by an example viewer with minimal dependencies
  - Building the sample applications now requires GCC 4.9 (previously 4.8) for features used in the C++ standard library; OSPRay itself can still be built with GCC 4.8
  - The new example viewer based on `ospray::sg` (called `ospExampleViewerSg`) is the single application we are consolidating to, `ospExampleViewer` will remain only as a deprecated viewer for compatibility with the old `ospGlutViewer` application
- Deprecated `ospCreateDevice()`; use `ospNewDevice()` instead
- Improved error handling
  - Various API functions now return an `OSPError` value
  - `ospDeviceSetStatusFunc` replaces the deprecated `ospDeviceSetErrorMsgFunc`
  - New API functions to query the last error (`ospDeviceGetLastErrorCode()` and `ospDeviceGetLastErrorMsg()`) or to register an error callback with `ospDeviceSetErrorFunc()`
  - Fixed bug where exceptions could leak to C applications

1.2.7 Changes in v1.2.1:

- Various bug fixes related to MPI distributed rendering, ISPC issues on Windows, and other build related issues

1.2.8 Changes in v1.2.0:

- Added support for volumes with voxelType `short` (16-bit signed integers). Applications need to recompile, because `OSPDataType` has been re-enumerated
- Removed SciVis renderer parameter `aoWeight`, the intensity (and now color as well) of AO is controlled via "`ambient`" lights. If `aoSamples` is zero (the default) then ambient lights cause ambient illumination (without occlusion)
- New SciVis renderer parameter `aoTransparencyEnabled`, controlling whether object transparency is respected when computing ambient occlusion (disabled by default, as it is considerable slower)
- Implement normal mapping for SciVis renderer
- Support of emissive (and illuminating) geometries in the path tracer via new material "`Luminous`"
- Lights can optionally made invisible by using the new parameter `isVisible` (only relevant for path tracer)
- OSPRay Devices are now extendable through modules and the SDK
  - Devices can be created and set current, creating an alternative method for initializing the API
  - New API functions for committing parameters on Devices
- Removed support for the first generation Intel® Xeon Phi™ coprocessor (codename Knights Corner)
- Other minor improvements, updates, and bug fixes
  - Updated Embree required version to v2.13.0 for added features and performance
  - New API function `ospDeviceSetErrorMsgFunc()` to specify a callback for handling message outputs from OSPRay
  - Added ability to remove user set parameter values with new `ospRemoveParam()` API function
  - The MPI device is now provided via a module, removing the need for having separately compiled versions of OSPRay with and without MPI
  - OSPRay build dependencies now only get installed if `OSPRAY_INSTALL_DEPENDENCIES` CMake variable is enabled

1.2.9 Changes in v1.1.2:

- Various bug fixes related to normalization, epsilons and debug messages

1.2.10 Changes in v1.1.1:

- Fixed support of first generation Intel Xeon Phi coprocessor (codename Knights Corner) and the COI device
- Fix normalization bug that caused rendering artifacts

1.2.11 Changes in v1.1.0:

- New "scivis" renderer features
  - Single sided lighting (enabled by default)

  - Many new volume rendering specific features
    * Adaptive sampling to help improve the correctness of rendering high frequency volume data
    * Pre-integration of transfer function for higher fidelity images
    * Ambient occlusion
    * Volumes can cast shadows
    * Smooth shading in volumes
    * Single shading point option for accelerated shading

- Added preliminary support for adaptive accumulation in the MPI device
- Camera specific features
  - Initial support for stereo rendering with the perspective camera
  - Option `architectural` in perspective camera, rectifying vertical edges to appear parallel
  - Rendering a subsection of the full view with `imageStart`/`imageEnd` supported by all cameras
- This will be our last release supporting the first generation Intel Xeon Phi coprocessor (codename Knights Corner)
  - Future major and minor releases will be upgraded to the latest version of Embree, which no longer supports Knights Corner
  - Depending on user feedback, patch releases are still made to fix bugs
- Enhanced output statistics in `ospBenchmark` application
- Many fixes to the OSPRay SDK
  - Improved CMake detection of compile-time enabled features
  - Now distribute OSPRay configuration and ISPC CMake macros
  - Improved SDK support on Windows
- OSPRay library can now be compiled with `-Wall` and `-Wextra` enabled
  - Tested with GCC v5.3.1 and Clang v3.8
  - Sample applications and modules have not been fixed yet, thus applications which build OSPRay as a CMake subproject should disable them with `-DOSPRAY_ENABLE_APPS=OFF` and `-DOSPRAY_ENABLE_MODULES=OFF`
- Minor bug fixes, improvements, and cleanups
  - Regard shading normal when bump mapping
  - Fix internal CMake naming inconsistencies in macros
  - Fix missing API calls in C++ wrapper classes
  - Fix crashes on MIC
  - Fix thread count initialization bug with TBB
  - CMake optimizations for faster configuration times
  - Enhanced support for scripting in both `ospGlutViewer` and `ospBenchmark` applications

1.2.12 Changes in v1.0.0:

- New OSPRay SDK
  - OSPRay internal headers are now installed, enabling applications to extend OSPRay from a binary install
  - CMake macros for OSPRay and ISPC configuration now a part of binary releases
    * CMake clients use them by calling `include(${OSPRAY_USE_FILE})` in their CMake code after calling `find_package(ospray)`

  - New OSPRay C++ wrapper classes
    * These act as a thin layer on top of OSPRay object handles, where multiple wrappers will share the same underlying handle when assigned, copied, or moved
    * New OSPRay objects are only created when a class instance is explicitly constructed
    * C++ users are encouraged to use these over the `ospray.h` API

- Complete rework of sample applications
  - New shared code for parsing the `commandline`
  - Save/load of transfer functions now handled through a separate library which does not depend on Qt
  - Added `ospCvtParaViewTfcn` utility, which enables `ospVolumeViewer` to load color maps from ParaView
  - GLUT based sample viewer updates
    * Rename of `ospModelViewer` to `ospGlutViewer`
    * GLUT viewer now supports volume rendering
    * Command mode with preliminary scripting capabilities, enabled by pressing '`:`' key (not available when using Intel C++ Compiler (icc))
  - Enhanced support of sample applications on Windows
- New minimum ISPC version is 1.9.0
- Support of Intel® AVX-512 for second generation Intel Xeon Phi processor (codename Knights Landing) is now a part of the `OSPRAY_BUILD_ISA` CMake build configuration
  - Compiling AVX-512 requires icc to be enabled as a build option
- Enhanced error messages when `ospLoadModule()` fails
- Added `OSP_FB_RGBA32F` support in the `DistributedFrameBuffer`
- Updated Glass shader in the path tracer
- Many miscellaneous cleanups, bug fixes, and improvements

1.2.13 Changes in v0.10.1:

- Fixed support of first generation Intel Xeon Phi coprocessor (codename Knights Corner)
- Restored missing implementation of `ospRemoveVolume()`

1.2.14 Changes in v0.10.0:

- Added new tasking options: `Cilk`, `Internal`, and `Debug`
  - Provides more ways for OSPRay to interact with calling application tasking systems
    * `Cilk`: Use Intel® Cilk™ Plus language extensions (icc only)
    * `Internal`: Use hand written OSPRay tasking system
    * `Debug`: All tasks are run in serial (useful for debugging)
  - In most cases, Intel Threading Building Blocks (Intel `TBB`) remains the fastest option
- Added support for adaptive accumulation and stopping
  - `ospRenderFrame` now returns an estimation of the variance in the rendered image if the framebuffer was created with the `OSP_FB_VARIANCE` channel

    - If the renderer parameter `varianceThreshold` is set, progressive refinement concentrates on regions of the image with a variance higher than this threshold

- Added support for volumes with voxelType `ushort` (16-bit unsigned integers)
- `OSPTexture2D` now supports sRGB formats – actually most images are stored in sRGB. As a consequence the API call `ospNewTexture2D()` needed to change to accept the new `OSPTextureFormat` parameter
- Similarly, OSPRay's framebuffer types now also distinguishes between linear and sRGB 8-bit formats. The new types are `OSP_FB_NONE`, `OSP_FB_RGBA8`, `OSP_FB_SRGBA`, and `OSP_FB_RGBA32F`
- Changed "scivis" renderer parameter defaults

    - All shading (AO + shadows) must be explicitly enabled

- OSPRay can now use a newer, pre-installed Embree enabled by the new `OSPRAY_USE_EXTERNAL_EMBREE` CMake option
- New `ospcommon` library used to separately provide math types and OS abstractions for both OSPRay and sample applications

    - Removes extra dependencies on internal Embree math types and utility functions
    - `ospray.h` header is now C99 compatible

- Removed loaders module, functionality remains inside of `ospVolumeViewer`
- Many miscellaneous cleanups, bug fixes, and improvements

    - Fixed data distributed volume rendering bugs when using less blocks than workers
    - Fixes to CMake `find_package()` config
    - Fix bug in `GhostBlockBrickVolume` when using `doubles`
    - Various robustness changes made in CMake to make it easier to compile OSPRay

1.2.15 Changes in v0.9.1:

- Volume rendering now integrated into the "scivis" renderer

    - Volumes are rendered in the same way the "dvr" volume renderer renders them
    - Ambient occlusion works with implicit isosurfaces, with a known visual quality/performance trade-off

- Intel Xeon Phi coprocessor (codename Knights Corner) COI device and build infrastructure restored (volume rendering is known to still be broken)
- New support for CPack built OSPRay binary redistributable packages
- Added support for HDRI lighting in path tracer
- Added `ospRemoveVolume()` API call
- Added ability to render a subsection of the full view into the entire framebuffer in the perspective camera
- Many miscellaneous cleanups, bug fixes, and improvements

    - The depthbuffer is now correctly populated by in the "scivis" renderer
    - Updated default renderer to be "ao1" in ospModelViewer
    - Trianglemesh `postIntersect` shading is now 64-bit safe
    - Texture2D has been reworked, with many improvements and bug fixes
    - Fixed bug where MPI device would freeze while rendering frames with Intel TBB
    - Updates to CMake with better error messages when Intel TBB is missing

1.2.16 Changes in v0.9.0:

The OSPRay v0.9.0 release adds significant new features as well as API changes.

- Experimental support for data-distributed MPI-parallel volume rendering
- New SciVis-focused renderer ("raytracer" or "scivis") combining functionality of "obj" and "ao" renderers
  - Ambient occlusion is quite flexible: dynamic number of samples, maximum ray distance, and weight
- Updated Embree version to v2.7.1 with native support for Intel AVX-512 for triangle mesh surface rendering on the Intel Xeon Phi processor (codename Knights Landing)
- OSPRay now uses C++11 features, requiring up to date compiler and standard library versions (GCC v4.8.0)
- Optimization of volume sampling resulting in volume rendering speedups of up to 1.5x
- Updates to path tracer
  - Reworked material system
  - Added texture transformations and colored transparency in OBJ material
  - Support for alpha and depth components of framebuffer
- Added thinlens camera, i.e. support for depth of field
- Tasking system has been updated to use Intel Threading Building Blocks (Intel TBB)
- The `ospGet*()` API calls have been deprecated and will be removed in a subsequent release

1.2.17 Changes in v0.8.3:

- Enhancements and optimizations to path tracer
  - Soft shadows (light sources: sphere, cone, extended spot, quad)
  - Transparent shadows
  - Normal mapping (OBJ material)
- Volume rendering enhancements
  - Expanded material support
  - Support for multiple lights
  - Support for double precision volumes
  - Added `ospSampleVolume()` API call to support limited probing of volume values
- New features to support compositing externally rendered content with OSPRay-rendered content
  - Renderers support early ray termination through a maximum depth parameter
  - New OpenGL utility module to convert between OSPRay and OpenGL depth values
- Added panoramic and orthographic camera types
- Proper CMake-based installation of OSPRay and CMake `find_package()` support for use in external projects
- Experimental Windows support
- Deprecated `ospNewTriangleMesh()`; use `ospNewGeometry("triangles")` instead
- Bug fixes and cleanups throughout the codebase

1.2.18 Changes in v0.8.2:

- Initial support for Intel AVX-512 and the Intel Xeon Phi processor (codename Knights Landing)
- Performance improvements to the volume renderer
- Incorporated implicit slices and isosurfaces of volumes as core geometry types
- Added support for multiple disjoint volumes to the volume renderer
- Improved performance of `ospSetRegion()`, reducing volume load times
- Improved large data handling for the `shared_structured_volume` and `block_bricked_volume` volume types
- Added support for DDS horizon data to the seismic module
- Initial support in the Qt viewer for volume rendering
- Updated to ISPC 1.8.2
- Various bug fixes, cleanups and documentation updates throughout the codebase

1.2.19 Changes in v0.8.1:

- The volume renderer and volume viewer can now be run MPI parallel (data replicated) using the `--osp:mpi` command line option
- Improved performance of volume grid accelerator generation, reducing load times for large volumes
- The volume renderer and volume viewer now properly handle multiple isosurfaces
- Added small example tutorial demonstrating how to use OSPRay
- Several fixes to support older versions of GCC
- Bug fixes to `ospSetRegion()` implementation for arbitrarily shaped regions and setting large volumes in a single call
- Bug fix for geometries with invalid bounds; fixes streamline and sphere rendering in some scenes
- Fixed bug in depth buffer generation

1.2.20 Changes in v0.8.0:

- Incorporated early version of a new Qt-based viewer to eventually unify (and replace) the existing simpler GLUT-based viewers
- Added new path tracing renderer (`ospray/render/pathtracer`),
- roughly based on the Embree sample path tracer
- Added new features to the volume renderer
  - Gradient shading (lighting)
  - Implicit isosurfacing
  - Progressive refinement
  - Support for regular grids, specified with the `gridOrigin` and `gridSpacing` parameters
  - New `shared_structured_volume` volume type that allows voxel data to be provided by applications through a shared data buffer
  - New API call to set subregions of volume data (`ospSetRegion()`)
- Added a subsampling-mode, enabled with a negative `spp` parameter; the first frame after scene changes is rendered with reduced resolution, increasing interactivity
- Added multi-target ISA support: OSPRay will now select the appropriate ISA at run time
- Added support for the Stanford SEP file format to the seismic module
- Added `--osp:numthreads <n>` command line option to restrict the number of threads OSPRay creates

- Various bug fixes, cleanups and documentation updates throughout the codebase

1.2.21 Changes in v0.7.2:

- Build fixes for older versions of GCC and Clang
- Fixed time series support in ospVolumeViewer
- Corrected memory management for shared data buffers
- Updated to ISPC 1.8.1
- Resolved issue in XML parser

Chapter 2

Building OSPRay from Source

The latest OSPRay sources are always available at the OSPRay GitHub repository. The default `master` branch should always point to the latest tested bugfix release.

2.1 Prerequisites

OSPRay currently supports Linux, Mac OS X, and Windows. In addition, before you can build OSPRay you need the following prerequisites:

- You can clone the latest OSPRay sources via:

  ```
  git clone https://github.com/ospray/ospray.git
  ```

- To build OSPRay you need CMake, any form of C++11 compiler (we recommend using GCC, but also support Clang and the Intel® C++ Compiler (icc)), and standard Linux development tools. To build the example viewers, you should also have some version of OpenGL.
- Additionally you require a copy of the Intel® SPMD Program Compiler (ISPC), version 1.9.1 or later. Please obtain a release of ISPC from the ISPC downloads page. The build system looks for ISPC in the `PATH` and in the directory right "next to" the checked-out OSPRay sources.[1] Alternatively set the CMake variable `ISPC_EXECUTABLE` to the location of the ISPC compiler.
- Per default OSPRay uses the Intel® Threading Building Blocks (TBB) as tasking system, which we recommend for performance and flexibility reasons. Alternatively you can set CMake variable `OSPRAY_TASKING_SYSTEM` to `OpenMP`, `Internal`, or `Cilk` (icc only).
- OSPRay also heavily uses Embree, installing version 2.15 or newer is required. If Embree is not found by CMake its location can be hinted with the variable `embree_DIR`.

[1] For example, if OSPRay is in `~/Projects/ospray`, ISPC will also be searched in `~/Projects/ispc-v1.9.2-linux`

Depending on your Linux distribution you can install these dependencies using `yum` or `apt-get`. Some of these packages might already be installed or might have slightly different names.

Type the following to install the dependencies using `yum`:

```
sudo yum install cmake.x86_64
sudo yum install tbb.x86_64 tbb-devel.x86_64
```

Type the following to install the dependencies using `apt-get`:

```
sudo apt-get install cmake-curses-gui
sudo apt-get install libtbb-dev
```

Under Mac OS X these dependencies can be installed using MacPorts:

```
sudo port install cmake tbb
```

Under Windows please directly use the appropriate installers for CMake, TBB, ISPC (for your Visual Studio version) and Embree.

2.2 Compiling OSPRay on Linux and Mac OS X

Assume the above requisites are all fulfilled, building OSPRay through CMake is easy:

- Create a build directory, and go into it

  ```
  mkdir ospray/build
  cd ospray/build
  ```

  (We do recommend having separate build directories for different configurations such as release, debug, etc).

- The compiler CMake will use will default to whatever the `CC` and `CXX` environment variables point to. Should you want to specify a different compiler, run cmake manually while specifying the desired compiler. The default compiler on most linux machines is `gcc`, but it can be pointed to `clang` instead by executing the following:

  ```
  cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang ..
  ```

  CMake will now use Clang instead of GCC. If you are ok with using the default compiler on your system, then simply skip this step. Note that the compiler variables cannot be changed after the first `cmake` or `ccmake` run.

- Open the CMake configuration dialog

  ```
  ccmake ..
  ```

- Make sure to properly set build mode and enable the components you need, etc; then type 'c'onfigure and 'g'enerate. When back on the command prompt, build it using

  ```
  make
  ```

- You should now have `libospray.so` as well as a set of example application. You can test your version of OSPRay using any of the examples on the OSPRay Demos and Examples page.

2.3 Compiling OSPRay on Windows

On Windows using the CMake GUI (`cmake-gui.exe`) is the most convenient way to configure OSPRay and to create the Visual Studio solution files:

- Browse to the OSPRay sources and specify a build directory (if it does not exist yet CMake will create it).
- Click "Configure" and select as generator the Visual Studio version you have, for Win64 (32 bit builds are not supported by OSPRay), e.g. "Visual Studio 15 2017 Win64".
- If the configuration fails because some dependencies could not be found then follow the instructions given in the error message, e.g. set the variable `embree_DIR` to the folder where Embree was installed.
- Optionally change the default build options, and then click "Generate" to create the solution and project files in the build directory.
- Open the generated `OSPRay.sln` in Visual Studio, select the build configuration and compile the project.

Alternatively, OSPRay can also be built without any GUI, entirely on the console. In the Visual Studio command prompt type:

```
cd path\to\ospray
mkdir build
cd build
cmake -G "Visual Studio 12 2013 Win64" [-D VARIABLE=value] ..
cmake --build . --config Release
```

Use `-D` to set variables for CMake, e.g. the path to Embree with "`-D embree_DIR=\path\to\embree`".

You can also build only some projects with the `--target` switch. Additional parameters after "`--`" will be passed to `msbuild`. For example, to build in parallel only the OSPRay library without the example applications use

```
cmake --build . --config Release --target ospray -- /m
```

Chapter 3
OSPRay API

To access the OSPRay API you first need to include the OSPRay header

```
#include "ospray/ospray.h"
```

where the API is compatible with C99 and C++.

3.1 Initialization

In order to use the API, OSPRay must be initialized with a "device". A device is the object which implements the API. Creating and initializing a device can be done in either of two ways: command line arguments or manually instantiating a device.

3.1.1 Command Line Arguments

The first is to do so by giving OSPRay the command line from `main()` by calling

```
OSPError ospInit(int *argc, const char **argv);
```

OSPRay parses (and removes) its known command line parameters from your application's `main` function. For an example see the tutorial. For possible error codes see section Error Handling and Status Messages. It is important to note that the arguments passed to `ospInit()` are processed in order they are listed. The following parameters (which are prefixed by convention with "`--osp:`") are understood:

3.1.2 Manual Device Instantiation

The second method of initialization is to explicitly create the device yourself, and possibly set parameters. This method looks almost identical to how other objects are created and used by OSPRay (described in later sections). The first step is to create the device with

```
OSPDevice ospNewDevice(const char *type);
```

where the `type` string maps to a specific device implementation. OSPRay always provides the "`default`" device, which maps to a local CPU rendering device. If it is enabled in the build, you can also use "`mpi`" to access the MPI multi-node rendering device (see Parallel Rendering with MPI section for more information). Once a device is created, you can call

```
void ospDeviceSet1i(OSPDevice, const char *id, int val);
```

Table 3.1 – Command line parameters accepted by OSPRay's `ospInit`.

Parameter	Description
`--osp:debug`	enables various extra checks and debug output, and disables multi-threading
`--osp:numthreads <n>`	use `n` threads instead of per default using all detected hardware threads
`--osp:loglevel <n>`	set logging level, default `0`; increasing `n` means increasingly verbose log messages
`--osp:verbose`	shortcut for `--osp:loglevel 1`
`--osp:vv`	shortcut for `--osp:loglevel 2`
`--osp:module:<name>`	load a module during initialization; equivalent to calling `ospLoadModule(name)`
`--osp:mpi`	enables MPI mode for parallel rendering with the `mpi_offload` device, to be used in conjunction with `mpirun`; this will automatically load the "mpi" module if it is not yet loaded or linked
`--osp:mpi-offload`	same as `--osp:mpi`
`--osp:mpi-distributed`	same as `--osp:mpi`, but will create an `mpi_distributed` device instead; Note that this will likely require application changes to work properly
`--osp:logoutput <dst>`	convenience for setting where status messages go; valid values for `dst` are `cerr` and `cout`
`--osp:erroroutput <dst>`	convenience for setting where error messages go; valid values for `dst` are `cerr` and `cout`
`--osp:device:<name>`	use `name` as the type of device for OSPRay to create; e.g. `--osp:device:default` gives you the default local device; Note if the device to be used is defined in a module, remember to pass `--osp:module:<name>` first
`--osp:setaffinity <n>`	if `1`, bind software threads to hardware threads; `0` disables binding; default is `1` on KNL and `0` otherwise

or

```
void ospDeviceSetString(OSPDevice, const char *id, const char *val);
```

to set parameters on the device. The following parameters can be set on all devices:

Table 3.2 – Parameters shared by all devices.

Type	Name	Description
int	numThreads	number of threads which OSPRay should use
int	logLevel	logging level
string	logOutput	convenience for setting where status messages go; valid values are `cerr` and `cout`
string	errorOutput	convenience for setting where error messages go; valid values are `cerr` and `cout`
int	debug	set debug mode; equivalent to logLevel=2 and numThreads=1
int	setAffinity	bind software threads to hardware threads if set to 1; 0 disables binding omitting the parameter will let OSPRay choose

Once parameters are set on the created device, the device must be committed with

```
void ospDeviceCommit(OSPDevice);
```

To use the newly committed device, you must call

```
void ospSetCurrentDevice(OSPDevice);
```

This then sets the given device as the object which will respond to all other OSPRay API calls.

Users can change parameters on the device after initialization (from either method above), by calling

```
OSPDevice ospGetCurrentDevice();
```

This function returns the handle to the device currently used to respond to OSPRay API calls, where users can set/change parameters and recommit the device. If changes are made to the device that is already set as the current device, it does not need to be set as current again.

3.1.3 Environment Variables

Finally, OSPRay's generic device parameters can be overridden via environment variables for easy changes to OSPRay's behavior without needing to change the application (variables are prefixed by convention with "`OSPRAY_`"):

Variable	Description
OSPRAY_THREADS	equivalent to `--osp:numthreads`
OSPRAY_LOG_LEVEL	equivalent to `--osp:loglevel`
OSPRAY_LOG_OUTPUT	equivalent to `--osp:logoutput`
OSPRAY_ERROR_OUTPUT	equivalent to `--osp:erroroutput`
OSPRAY_DEBUG	equivalent to `--osp:debug`
OSPRAY_SET_AFFINITY	equivalent to `--osp:setaffinity`

Table 3.3 – Environment variables interpreted by OSPRay.

3.1.4 Error Handling and Status Messages

The following errors are currently used by OSPRay:

Name	Description
OSP_NO_ERROR	no error occurred
OSP_UNKNOWN_ERROR	an unknown error occurred
OSP_INVALID_ARGUMENT	an invalid argument was specified
OSP_INVALID_OPERATION	the operation is not allowed for the specified object
OSP_OUT_OF_MEMORY	there is not enough memory to execute the command
OSP_UNSUPPORTED_CPU	the CPU is not supported (minimum ISA is SSE4.1)

Table 3.4 – Possible error codes, i.e. valid named constants of type `OSPError`.

These error codes are either directly return by some API functions, or are recorded to be later queried by the application via

```
OSPError ospDeviceGetLastErrorCode(OSPDevice);
```

A more descriptive error message can be queried by calling

```
const char* ospDeviceGetLastErrorMsg(OSPDevice);
```

Alternatively, the application can also register a callback function of type

```
typedef void (*OSPErrorFunc)(OSPError, const char* errorDetails);
```

via

```
void ospDeviceSetErrorFunc(OSPDevice, OSPErrorFunc);
```

to get notified when errors occur.

Applications may be interested in messages which OSPRay emits, whether for debugging or logging events. Applications can call

```
void ospDeviceSetStatusFunc(OSPDevice, OSPStatusFunc);
```

in order to register a callback function of type

```
typedef void (*OSPStatusFunc)(const char* messageText);
```

which OSPRay will use to emit status messages. By default, OSPRay uses a callback which does nothing, so any output desired by an application will require that a callback is provided. Note that callbacks for C++ `std::cout` and `std::cerr` can be alternatively set through `ospInit()` or the `OSPRAY_LOG_OUTPUT` environment variable.

3.1.5 Loading OSPRay Extensions at Runtime

OSPRay's functionality can be extended via plugins, which are implemented in shared libraries. To load plugin `name` from `libospray_module_<name>.so` (on Linux and Mac OS X) or `ospray_module_<name>.dll` (on Windows) use

```
OSPError ospLoadModule(const char *name);
```

Modules are searched in OS-dependent paths. `ospLoadModule` returns `OSP_NO_ERROR` if the plugin could be successfully loaded.

3.2 Objects

All entities of OSPRay (the renderer, volumes, geometries, lights, cameras, ...) are a specialization of `OSPObject` and share common mechanism to deal with parameters and lifetime.

An important aspect of object parameters is that parameters do not get passed to objects immediately. Instead, parameters are not visible at all to objects until they get explicitly committed to a given object via a call to

```
void ospCommit(OSPObject);
```

at which time all previously additions or changes to parameters are visible at the same time. If a user wants to change the state of an existing object (e.g., to change the origin of an already existing camera) it is perfectly valid to do so, as long as the changed parameters are recommitted.

The commit semantic allow for batching up multiple small changes, and specifies exactly when changes to objects will occur. This is important to ensure performance and consistency for devices crossing a PCI bus, or across a network. In our MPI implementation, for example, we can easily guarantee consistency among different nodes by MPI barrier'ing on every commit.

Note that OSPRay uses reference counting to manage the lifetime of all objects, so one cannot explicitly "delete" any object. Instead, to indicate that the application does not need and does not access the given object anymore, call

```
void ospRelease(OSPObject);
```

This decreases its reference count and if the count reaches `0` the object will automatically get deleted.

3.2.1 Parameters

Parameters allow to configure the behavior of and to pass data to objects. However, objects do *not* have an explicit interface for reasons of high flexibility and a more stable compile-time API. Instead, parameters are passed separately to objects in an arbitrary order, and unknown parameters will simply be ignored. The following functions allow adding various types of parameters with name `id` to a given object:

```
// add a C-string (zero-terminated char *) parameter
void ospSetString(OSPObject, const char *id, const char *s);

// add an object handle parameter to another object
void ospSetObject(OSPObject, const char *id, OSPObject object);

// add an untyped pointer -- this will *ONLY* work in local rendering!
void ospSetVoidPtr(OSPObject, const char *id, void *v);

// add scalar and vector integer and float parameters
void ospSetf  (OSPObject, const char *id, float x);
void ospSet1f (OSPObject, const char *id, float x);
void ospSet1i (OSPObject, const char *id, int32_t x);
void ospSet2f (OSPObject, const char *id, float x, float y);
void ospSet2fv(OSPObject, const char *id, const float *xy);
void ospSet2i (OSPObject, const char *id, int x, int y);
void ospSet2iv(OSPObject, const char *id, const int *xy);
void ospSet3f (OSPObject, const char *id, float x, float y, float z);
void ospSet3fv(OSPObject, const char *id, const float *xyz);
void ospSet3i (OSPObject, const char *id, int x, int y, int z);
void ospSet3iv(OSPObject, const char *id, const int *xyz);
void ospSet4f (OSPObject, const char *id, float x, float y, float z, float w);
void ospSet4fv(OSPObject, const char *id, const float *xyzw);

// additional functions to pass vector integer and float parameters in C++
void ospSetVec2f(OSPObject, const char *id, const vec2f &v);
void ospSetVec2i(OSPObject, const char *id, const vec2i &v);
void ospSetVec3f(OSPObject, const char *id, const vec3f &v);
void ospSetVec3i(OSPObject, const char *id, const vec3i &v);
void ospSetVec4f(OSPObject, const char *id, const vec4f &v);
```

Users can also remove parameters that have been explicitly set via an ospSet call. Any parameters which have been removed will go back to their default value during the next commit unless a new parameter was set after the parameter was removed. The following API function removes the named parameter from the given object:

```
void ospRemoveParam(OSPObject, const char *id);
```

3.2.2 Data

There is also the possibility to aggregate many values of the same type into an array, which then itself can be used as a parameter to objects. To create such a new data buffer, holding `numItems` elements of the given type, from the initialization data pointed to by `source` and optional creation flags, use

```
OSPData ospNewData(size_t numItems,
                   OSPDataType,
                   const void *source,
                   const uint32_t dataCreationFlags = 0);
```

The call returns an `OSPData` handle to the created array. The flag `OSP_DATA_SHARED_BUFFER` indicates that the buffer can be shared with the application. In this case the calling program guarantees that the `source` pointer will remain valid for the duration that this data array is being used. The enum type `OSPDataType` describes the different data types that can be represented in OSPRay; valid constants are listed in the table below.

Type/Name	Description
OSP_DEVICE	API device object reference
OSP_VOID_PTR	void pointer
OSP_DATA	data reference
OSP_OBJECT	generic object reference
OSP_CAMERA	camera object reference
OSP_FRAMEBUFFER	framebuffer object reference
OSP_LIGHT	light object reference
OSP_MATERIAL	material object reference
OSP_TEXTURE	texture object reference
OSP_RENDERER	renderer object reference
OSP_MODEL	model object reference
OSP_GEOMETRY	geometry object reference
OSP_VOLUME	volume object reference
OSP_TRANSFER_FUNCTION	transfer function object reference
OSP_PIXEL_OP	pixel operation object reference
OSP_STRING	C-style zero-terminated character string
OSP_CHAR	8 bit signed character scalar
OSP_UCHAR	8 bit unsigned character scalar
OSP_UCHAR[234]	… and [234]-element vector
OSP_USHORT	16 bit unsigned integer scalar
OSP_INT	32 bit signed integer scalar
OSP_INT[234]	… and [234]-element vector
OSP_UINT	32 bit unsigned integer scalar
OSP_UINT[234]	… and [234]-element vector
OSP_LONG	64 bit signed integer scalar
OSP_LONG[234]	… and [234]-element vector
OSP_ULONG	64 bit unsigned integer scalar
OSP_ULONG[234]	… and [234]-element vector
OSP_FLOAT	32 bit single precision floating point scalar
OSP_FLOAT[234]	… and [234]-element vector
OSP_FLOAT3A	… and aligned 3-element vector
OSP_DOUBLE	64 bit double precision floating point scalar

Table 3.5 – Valid named constants for `OSPDataType`.

To add a data array as parameter named `id` to another object call

```
void ospSetData(OSPObject, const char *id, OSPData);
```

3.3 Volumes

Volumes are volumetric datasets with discretely sampled values in 3D space, typically a 3D scalar field. To create a new volume object of given type `type` use

```
OSPVolume ospNewVolume(const char *type);
```

The call returns `NULL` if that type of volume is not known by OSPRay, or else an `OSPVolume` handle.

The common parameters understood by all volume variants are summarized in the table below.

Table 3.6 – Configuration parameters shared by all volume types.

Type	Name	Default	Description
vec2f	voxelRange		minimum and maximum of the scalar values
bool	gradientShadingEnabled	false	volume is rendered with surface shading wrt. to normalized gradient
bool	preIntegration	false	use pre-integration for transfer function lookups
bool	singleShade	true	shade only at the point of maximum intensity
bool	adaptiveSampling	true	adapt ray step size based on opacity
float	adaptiveScalar	15	modifier for adaptive step size
float	adaptiveMaxSamplingRate	2	maximum sampling rate for adaptive sampling
float	samplingRate	0.125	sampling rate of the volume (this is the minimum step size for adaptive sampling)
vec3f	specular	gray 0.3	specular color for shading
vec3f	volumeClippingBoxLower	disabled	lower coordinate (in object-space) to clip the volume values
vec3f	volumeClippingBoxUpper	disabled	upper coordinate (in object-space) to clip the volume values

Note that if `voxelRange` is not provided for a volume then OSPRay will compute it based on the voxel data, which may result in slower data updates.

3.3.1 Structured Volume

Structured volumes only need to store the values of the samples, because their addresses in memory can be easily computed from a 3D position. A common type of structured volumes are regular grids. OSPRay supports two variants that differ in how the volumetric data for the regular grids is specified.

The first variant shares the voxel data with the application. Such a volume type is created by passing the type string "`shared_structured_volume`" to `ospNewVolume`. The voxel data is laid out in memory in XYZ order and provided to the volume via a data buffer parameter named "`voxelData`".

The second regular grid variant is optimized for rendering performance: data locality in memory is increased by arranging the voxel data in smaller blocks. This volume type is created by passing the type string "`block_bricked_volume`" to `ospNewVolume`. Because of this rearrangement of voxel data it cannot be shared the with the application anymore, but has to be transferred to OSPRay via

```
OSPError ospSetRegion(OSPVolume, void *source,
                      const vec3i &regionCoords,
                      const vec3i &regionSize);
```

The voxel data pointed to by `source` is copied into the given volume starting at position `regionCoords`, must be of size `regionSize` and be placed in memory in XYZ order. Note that OSPRay distinguishes between volume data and volume parameters. This function must be called only after all volume parameters (in particular `dimensions` and `voxelType`, see below) have been set and *before* `ospCommit(volume)` is called. If necessary then memory for the volume is allocated on the first call to this function.

The common parameters understood by both structured volume variants are summarized in the table below.

Table 3.7 – Additional configuration parameters for structured volumes.

Type	Name	Default	Description
vec3i	dimensions		number of voxels in each dimension (x, y, z)
string	voxelType		data type of each voxel, currently supported are:
			"uchar" (8 bit unsigned integer)
			"short" (16 bit signed integer)
			"ushort" (16 bit unsigned integer)
			"float" (32 bit single precision floating point)
			"double" (64 bit double precision floating point)
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in world-space
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in world-space

3.3.2 Adaptive Mesh Refinement (AMR) Volume

AMR volumes are specified as a list of bricks, which are levels of refinement in potentially overlapping regions. There can be any number of refinement levels and any number of bricks at any level of refinement. An AMR volume type is created by passing the type string "`amr_volume`" to `ospNewVolume`.

Applications should first create an `OSPData` array which holds information about each brick. The following structure is used to populate this array (found in `ospray.h`):

```
struct amr_brick_info
{
  box3i bounds;
  int   refinementLevel;
  float cellWidth;
};
```

Then for each brick, the application should create an `OSPData` array of `OSPData` handles, where each handle is the data per-brick. Currently we only support `float` voxels.

Lastly, note that the `gridOrigin` and `gridSpacing` parameters act just like the structured volume equivalent, but they only modify the root (coarsest level) of refinement.

3.3.3 Unstructured Tetrahedral Volumes

Unstructured tetrahedral volumes are defined by three arrays: vertices, corresponding field values, and tetrahedra indices. A tetrahedral volume type is created by passing the type string "`tetrahedral_volume`" to `ospNewVolume`.

Table 3.8 – Additional configuration parameters for AMR volumes.

Type	Name	Default	Description
vec3f	gridOrigin	$(0, 0, 0)$	origin of the grid in world-space
vec3f	gridSpacing	$(1, 1, 1)$	size of the grid cells in world-space
string	amrMethod	current	sampling method; valid values are “finest”, “current”, or “octant”
OSPData	brickInfo		array of info defining each brick
OSPData	brickData		array of handles to per-brick voxel data

Similar to triangle mesh, each tetrahedron is formed by a group of indices into the vertices. For each vertex, the corresponding (by array index) data value will be used for sampling when rendering. Note that the index order for each tetrahedron does not matter, as OSPRay internally calculates vertex normals to ensure proper sampling and interpolation.

Type	Name	Description
vec3f[]	vertices	data array of vertex positions
float[]	field	data array of vertex data values to be sampled
vec4i[]	tetrahedra	data array of tetrahedra indices (into vertices and field)

Table 3.9 – Additional configuration parameters for tetrahedral volumes.

3.3.4 Transfer Function

Transfer functions map the scalar values of volumes to color and opacity and thus they can be used to visually emphasize certain features of the volume. To create a new transfer function of given type `type` use

```
OSPTransferFunction ospNewTransferFunction(const char *type);
```

The call returns `NULL` if that type of transfer functions is not known by OSPRay, or else an `OSPTransferFunction` handle to the created transfer function. That handle can be assigned to a volume as parameter “`transferFunction`” using `ospSetObject`.

One type of transfer function that is built-in in OSPRay is the linear transfer function, which interpolates between given equidistant colors and opacities. It is create by passing the string “`piecewise_linear`” to `ospNewTransferFunction` and it is controlled by these parameters:

Type	Name	Description
vec3f[]	colors	data array of RGB colors
float[]	opacities	data array of opacities
vec2f	valueRange	domain (scalar range) this function maps from

Table 3.10 – Parameters accepted by the linear transfer function.

3.4 Geometries

Geometries in OSPRay are objects that describe surfaces. To create a new geometry object of given type `type` use

```
OSPGeometry ospNewGeometry(const char *type);
```

The call returns `NULL` if that type of geometry is not known by OSPRay, or else an `OSPGeometry` handle.

3.4.1 Triangle Mesh

A traditional triangle mesh (indexed face set) geometry is created by calling `ospNewGeometry` with type string "`triangles`". Once created, a triangle mesh recognizes the following parameters:

Type	Name	Description
vec3f(a)[]	vertex	data array of vertex positions
vec3f(a)[]	vertex.normal	data array of vertex normals
vec4f[] / vec3fa[]	vertex.color	data array of vertex colors (RGBA/RGB)
vec2f[]	vertex.texcoord	data array of vertex texture coordinates
vec3i(a)[]	index	data array of triangle indices (into vertex.*)

Table 3.11 – Parameters defining a triangle mesh geometry.

3.4.2 Spheres

A geometry consisting of individual spheres, each of which can have an own radius, is created by calling `ospNewGeometry` with type string "`spheres`". The spheres will not be tessellated but rendered procedurally and are thus perfectly round. To allow a variety of sphere representations in the application this geometry allows a flexible way of specifying the data of center position and radius within a data array:

Table 3.12 – Parameters defining a spheres geometry.

Type	Name	Default	Description
float	radius	0.01	radius of all spheres (if `offset_radius` is not used)
OSPData	spheres	NULL	memory holding the spatial data of all spheres
int	bytes_per_sphere	16	size (in bytes) of each sphere within the `spheres` array
int	offset_center	0	offset (in bytes) of each sphere's "vec3f center" position (in object-space) within the `spheres` array
int	offset_radius	-1	offset (in bytes) of each sphere's "float radius" within the `spheres` array (`-1` means disabled and use `radius`)
vec4f[] / vec3f(a)[]	color	NULL	data array of colors (RGBA/RGB), color is constant for each sphere
vec2f[]	texcoord	NULL	data array of texture coordinates, coordinate is constant for each sphere

3.4.3 Cylinders

A geometry consisting of individual cylinders, each of which can have an own radius, is created by calling `ospNewGeometry` with type string "`cylinders`". The cylinders will not be tessellated but rendered procedurally and are thus perfectly round. To allow a variety of cylinder representations in the application this geometry allows a flexible way of specifying the data of offsets for start position, end position and radius within a data array. All parameters are listed in the table below.

Table 3.13 – Parameters defining a cylinders geometry.

Type	Name	Default	Description
float	radius	0.01	radius of all cylinders (if `offset_radius` is not used)
OSPData	cylinders	NULL	memory holding the spatial data of all cylinders
int	bytes_per_cylinder	24	size (in bytes) of each cylinder within the `cylinders` array
int	offset_v0	0	offset (in bytes) of each cylinder's "vec3f v0" position (the start vertex, in object-space) within the `cylinders` array
int	offset_v1	12	offset (in bytes) of each cylinder's "vec3f v1" position (the end vertex, in object-space) within the `cylinders` array
int	offset_radius	-1	offset (in bytes) of each cylinder's "float radius" within the `cylinders` array (`-1` means disabled and use `radius` instead)
vec4f[] / vec3f(a)[]	color	NULL	data array of colors (RGBA/RGB), color is constant for each cylinder
OSPData	texcoord	NULL	data array of texture coordinates, in pairs (each a vec2f at vertex v0 and v1)

For texturing each cylinder is seen as a 1D primitive, i.e. a line segment: the 2D texture coordinates at its vertices v0 and v1 are linearly interpolated.

3.4.4 Streamlines

A geometry consisting of multiple streamlines of constant radius is created by calling `ospNewGeometry` with type string "`streamlines`". The streamlines are internally assembled from connected (and rounded) cylinder segments and are thus perfectly round. The parameters defining this geometry are listed in the table below.

Type	Name	Description
float	radius	radius of all streamlines, default 0.01
vec3fa[]	vertex	data array of all vertices for *all* streamlines
vec4f[]	vertex.color	data array of corresponding vertex colors (RGBA)
int32[]	index	data array of indices to the first vertex of a link

Table 3.14 – Parameters defining a streamlines geometry.

Each streamline is specified by a set of (aligned) vec3fa control points in `vertex`; all vertices belonging to to the same logical streamline are connected via cylinders of a fixed radius `radius`, with additional spheres at each vertex to make for a smooth transition between the cylinders.

A streamlines geometry can contain multiple disjoint streamlines, each streamline is specified as a list of linear segments (or links) referenced via `index`: each entry `e` of the `index` array points the first vertex of a link (`vertex[index[e]]`) and the second vertex of the link is implicitly the directly following one (`vertex[index[e]+1]`). For example, two streamlines of vertices (`A-B-C-D`) and (`E-F-G`), respectively, would internally correspond to five links (`A-B`, `B-C`, `C-D`, `E-F`, and `F-G`), and would be specified via an array of vertices `[A,B,C,D,E,F,G]`, plus an array of link indices `[0,1,2,4,5]`.

3.4.5 Isosurfaces

OSPRay can directly render multiple isosurfaces of a volume without first tesselating them. To do so create an isosurfaces geometry by calling `ospNewGeometry` with type string "`isosurfaces`". Each isosurface will be colored according to the provided volume's transfer function.

Type	Name	Description
float[]	isovalues	data array of isovalues
OSPVolume	volume	handle of the volume to be isosurfaced

Table 3.15 – Parameters defining an isosurfaces geometry.

3.4.6 Slices

One tool to highlight interesting features of volumetric data is to visualize 2D cuts (or slices) by placing planes into the volume. Such a slices geometry is created by calling `ospNewGeometry` with type string "`slices`". The planes are defined by the coefficients (a, b, c, d) of the plane equation $ax + by + cz + d = 0$. Each slice is colored according to the provided volume's transfer function.

Type	Name	Description
vec4f[]	planes	data array with plane coefficients for all slices
OSPVolume	volume	handle of the volume that will be sliced

Table 3.16 – Parameters defining a slices geometry.

3.4.7 Instances

OSPRay supports instancing via a special type of geometry. Instances are created by transforming another given model `modelToInstantiate` with the given affine transformation `transform` by calling

```
OSPGeometry ospNewInstance(OSPModel modelToInstantiate, const affine3f &transform);
```

3.5 Renderer

A renderer is the central object for rendering in OSPRay. Different renderers implement different features and support different materials. To create a new renderer of given type `type` use

```
OSPRenderer ospNewRenderer(const char *type);
```

The call returns `NULL` if that type of renderer is not known, or else an `OSPRenderer` handle to the created renderer. General parameters of all renderers are

OSPRay's renderers support a feature called adaptive accumulation, which accelerates progressive rendering by stopping the rendering and refinement of image regions that have an estimated variance below the `varianceThreshold`. This feature requires a framebuffer with an `OSP_FB_VARIANCE` channel.

Table 3.17 – Parameters understood by all renderers.

Type	Name	Default	Description
OSPModel	model		the model to render
OSPCamera	camera		the camera to be used for rendering
OSPLight[]	lights		data array with handles of the lights
float	epsilon	10^{-6}	ray epsilon to avoid self-intersections, relative to scene diameter
int	spp	1	samples per pixel
int	maxDepth	20	maximum ray recursion depth
float	minContribution	0.001	sample contributions below this value will be neglected to speed-up rendering
float	varianceThreshold	0	threshold for adaptive accumulation

3.5.1 SciVis Renderer

The SciVis renderer is a fast ray tracer for scientific visualization which supports volume rendering and ambient occlusion (AO). It is created by passing the type string “`scivis`” or “`raytracer`” to `ospNewRenderer`. In addition to the general parameters understood by all renderers the SciVis renderer supports the following special parameters:

Table 3.18 – Special parameters understood by the SciVis renderer.

Type	Name	Default	Description
bool	shadowsEnabled	false	whether to compute (hard) shadows
int	aoSamples	0	number of rays per sample to compute ambient occlusion
float	aoDistance	10^{20}	maximum distance to consider for ambient occlusion
bool	aoTransparencyEnabled	false	whether object transparency is respected when computing ambient occlusion (slower)
bool	oneSidedLighting	true	if true back-facing surfaces (wrt. light source) receive no illumination
float / vec3f / vec4f	bgColor	black, transparent	background color and alpha (RGBA)
OSPTexture2D	maxDepthTexture	NULL	screen-sized float texture with maximum far distance per pixel

Note that the intensity (and color) of AO is controlled via an ambient light. If `aoSamples` is zero (the default) then ambient lights cause ambient illumination (without occlusion).

Per default the background of the rendered image will be transparent black, i.e. the alpha channel holds the opacity of the rendered objects. This facilitates transparency-aware blending of the image with an arbitrary background image by the application. The parameter `bgColor` can be used to already blend with a constant background color (and alpha) during rendering.

The SciVis renderer supports depth composition with images of other renderers, for example to incorporate help geometries of a 3D UI that were rendered with OpenGL. The screen-sized texture `maxDepthTexture` must have format `OSP_TEXTURE_R32F` and flag `OSP_TEXTURE_FILTER_NEAREST`. The fetched values are used to limit the distance of primary rays, thus objects of other renderers can hide objects rendered by OSPRay.

3.5.2 Path Tracer

The path tracer supports soft shadows, indirect illumination and realistic materials. This renderer is created by passing the type string "`pathtracer`" to `ospNewRenderer`. In addition to the general parameters understood by all renderers the path tracer supports the following special parameters:

Table 3.19 – Special parameters understood by the path tracer.

Type	Name	Default	Description
int	rouletteDepth	5	ray recursion depth at which to start Russian roulette termination
float	maxContribution	∞	samples are clamped to this value before they are accumulated into the framebuffer
OSPTexture2D	backplate	NULL	texture image used as background, replacing visible lights in infinity (e.g. the HDRI light)

The path tracer requires that materials are assigned to geometries, otherwise surfaces are treated as completely black.

3.5.3 Model

Models are a container of scene data. They can hold the different geometries and volumes as well as references to (and instances of) other models. A model is associated with a single logical acceleration structure. To create an (empty) model call

```
OSPModel ospNewModel();
```

The call returns an `OSPModel` handle to the created model. To add an already created geometry or volume to a model use

```
void ospAddGeometry(OSPModel, OSPGeometry);
void ospAddVolume(OSPModel, OSPVolume);
```

An existing geometry or volume can be removed from a model with

```
void ospRemoveGeometry(OSPModel, OSPGeometry);
void ospRemoveVolume(OSPModel, OSPVolume);
```

3.5.4 Lights

To let the given `renderer` create a new light source of given type `type` use

```
OSPLight ospNewLight(OSPRenderer renderer, const char *type);
```

The call returns `NULL` if that type of camera is not known by the renderer, or else an `OSPLight` handle to the created light source. All light sources[1] accept the following parameters:

[1] The HDRI Light is an exception, it knows about `intensity`, but not about `color`.

Table 3.20 – Parameters accepted by the all lights.

Type	Name	Default	Description
vec3f(a)	color	white	color of the light
float	intensity	1	intensity of the light (a factor)
bool	isVisible	true	whether the light can be directly seen

The following light types are supported by most OSPRay renderers.

3.5.4.1 Directional Light / Distant Light

The distant light (or traditionally the directional light) is thought to be very far away (outside of the scene), thus its light arrives (almost) as parallel rays. It is created by passing the type string "`distant`" to `ospNewLight`. In addition to the general parameters understood by all lights the distant light supports the following special parameters:

Type	Name	Description
vec3f(a)	direction	main emission direction of the distant light
float	angularDiameter	apparent size (angle in degree) of the light

Table 3.21 – Special parameters accepted by the distant light.

Setting the angular diameter to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the path tracer). For instance, the apparent size of the sun is about 0.53°.

3.5.4.2 Point Light / Sphere Light

The sphere light (or the special case point light) is a light emitting uniformly in all directions. It is created by passing the type string "`sphere`" to `ospNewLight`. In addition to the general parameters understood by all lights the sphere light supports the following special parameters:

Type	Name	Description
vec3f(a)	position	the center of the sphere light, in world-space
float	radius	the size of the sphere light

Table 3.22 – Special parameters accepted by the sphere light.

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the path tracer).

3.5.4.3 Spot Light

The spot light is a light emitting into a cone of directions. It is created by passing the type string "`spot`" to `ospNewLight`. In addition to the general parameters understood by all lights the spot light supports the special parameters listed in the table.

Table 3.23 – Special parameters accepted by the spot light.

Type	Name	Description
vec3f(a)	position	the center of the spot light, in world-space
vec3f(a)	direction	main emission direction of the spot
float	openingAngle	full opening angle (in degree) of the spot; outside of this cone is no illumination
float	penumbraAngle	size (angle in degree) of the "penumbra", the region between the rim (of the illumination cone) and full intensity of the spot; should be smaller than half of `openingAngle`
float	radius	the size of the spot light, the radius of a disk with normal `direction`

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling (like the path tracer).

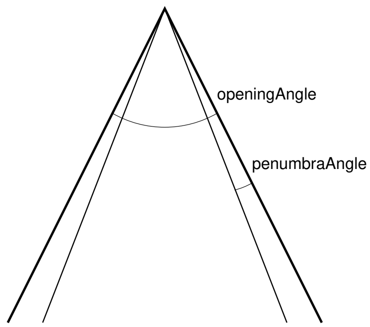

Figure 3.1 – Angles used by SpotLight.

3.5.4.4 Quad Light

The quad[2] light is a planar, procedural area light source emitting uniformly on one side into the half space. It is created by passing the type string "`quad`" to `ospNewLight`. In addition to the general parameters understood by all lights the spot light supports the following special parameters:

[2] actually a parallelogram

Type	Name	Description
vec3f(a)	position	world-space position of one vertex of the quad light
vec3f(a)	edge1	vector to one adjacent vertex
vec3f(a)	edge2	vector to the other adjacent vertex

Table 3.24 – Special parameters accepted by the quad light.

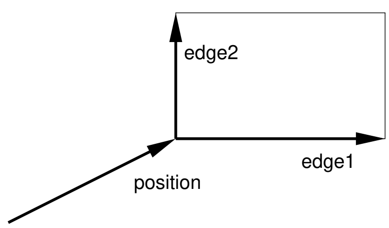

Figure 3.2 – Defining a Quad Light.

The emission side is determined by the cross product of `edge1`×`edge2`. Note that only renderers that use stochastic sampling (like the path tracer) will compute soft shadows from the quad light. Other renderers will just sample the center of the quad light, which results in hard shadows.

3.5.4.5 HDRI Light

The HDRI light is a textured light source surrounding the scene and illuminating it from infinity. It is created by passing the type string "`hdri`" to `ospNewLight`. In addition to the parameter `intensity` the HDRI light supports the following special parameters:

Table 3.25 – Special parameters accepted by the HDRI light.

Type	Name	Description
vec3f(a)	up	up direction of the light in world-space
vec3f(a)	dir	direction to which the center of the texture will be mapped to (analog to panoramic camera)
OSPTexture2D	map	environment map in latitude / longitude format

Note that the currently only the path tracer supports the HDRI light.

Figure 3.3 – Orientation and Mapping of an HDRI Light.

3.5.4.6 Ambient Light

The ambient light surrounds the scene and illuminates it from infinity with constant radiance (determined by combining the parameters `color` and `intensity`). It is created by passing the type string "`ambient`" to `ospNewLight`.

Note that the SciVis renderer uses ambient lights to control the color and intensity of the computed ambient occlusion (AO).

3.5.4.7 Emissive Objects

The path tracer will consider illumination by geometries which have a light emitting material assigned (for example the Luminous material).

3.5.5 Materials

Materials describe how light interacts with surfaces, they give objects their distinctive look. To let the given renderer create a new material of given type `type` call

```
OSPMaterial ospNewMaterial(OSPRenderer, const char *type);
```

The call returns `NULL` if the material type is not known by the renderer, or else an `OSPMaterial` handle to the created material. The handle can then be used to assign the material to a given geometry with

```
void ospSetMaterial(OSPGeometry, OSPMaterial);
```

3.5.5.1 OBJ Material

The OBJ material is the workhorse material supported by both the SciVis renderer and the path tracer. It offers widely used common properties like diffuse and specular reflection and is based on the MTL material format of Lightwave's OBJ scene files. To create an OBJ material pass the type string "`OBJMaterial`" to `ospNewMaterial`. Its main parameters are

Type	Name	Default	Description
vec3f	Kd	white 0.8	diffuse color
vec3f	Ks	black	specular color
float	Ns	10	shininess (Phong exponent), usually in [2–10^4]
float	d	opaque	opacity
vec3f	Tf	black	transparency filter color
OSPTexture2D	map_Bump	NULL	normal map

Table 3.26 – Main parameters of the OBJ material.

In particular when using the path tracer it is important to adhere to the principle of energy conservation, i.e. that the amount of light reflected by a surface is not larger than the light arriving. Therefore the path tracer issues a warning and renormalizes the color parameters if the sum of `Kd`, `Ks`, and `Tf` is larger than one in any color channel. Similarly important to mention is that almost all materials of the real world reflect at most only about 80% of the incoming light. So even for a white sheet of paper or white wall paint do better not set `Kd` larger than 0.8; otherwise rendering times are unnecessary long and the contrast in the final images is low (for example, the corners of a white room would hardly be discernible).

Note that currently only the path tracer implements colored transparency with `Tf`.

Normal mapping can simulate small geometric features via the texture `map_Bump`. The normals n in the normal map are wrt. the local tangential shading coordinate system and are encoded as $1/2(n+1)$, thus a texel $(0.5, 0.5, 1)$[3] represents the unperturbed shading normal $(0, 0, 1)$. Because of this encoding an sRGB gamma texture format is ignored and normals are always fetched as linear from a normal map. Note that the orientation of normal maps is important for a visually consistent look: by convention OSPRay uses a coordinate system with the origin in the lower left corner; thus a convexity will look green towards the top of the texture image (see also the example image of a normal map). If this is not the case flip the normal map vertically or invert its green channel.

[3] respectively $(127, 127, 255)$ for 8 bit textures

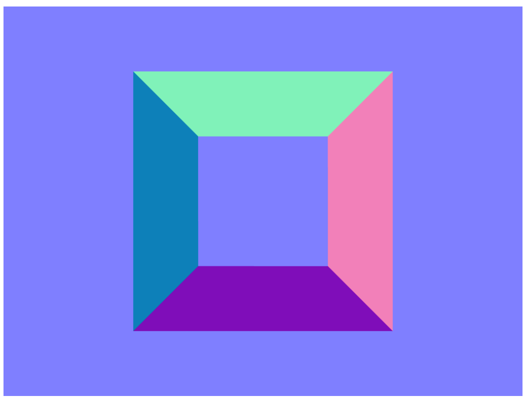

Figure 3.4 – Normal map representing an exalted square pyramidal frustum.

All parameters (except `Tf`) can be textured by passing a texture handle, prefixed with "`map_`". The fetched texels are multiplied by the respective parameter value. Texturing requires geometries with texture coordinates, e.g. a triangle mesh with `vertex.texcoord` provided. The color textures `map_Kd` and `map_Ks` are typically in one of the sRGB gamma encoded formats, whereas textures `map_Ns` and `map_d` are usually in a linear format (and only the first component is used). Additionally, all textures support texture transformations.

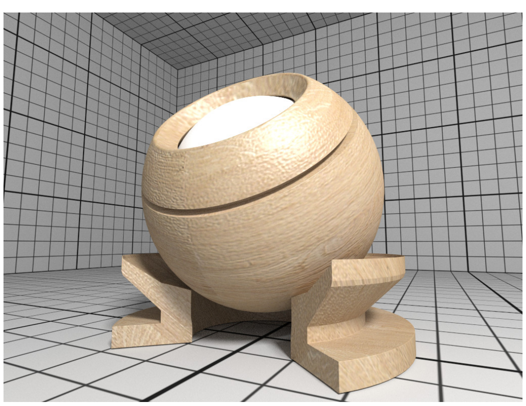

Figure 3.5 – Rendering of a OBJ material with wood textures.

3.5.5.2 Metal

The path tracer offers a physical metal, supporting changing roughness and realistic color shifts at edges. To create a Metal material pass the type string "`Metal`" to `ospNewMaterial`. Its parameters are

Table 3.27 – Parameters of the Metal material.

Type	Name	Default	Description
vec3f[]	ior	Aluminium	data array of spectral samples of complex refractive index, each entry in the form (wavelength, eta, k), ordered by wavelength (which is in nm)
vec3f	eta		RGB complex refractive index, real part
vec3f	k		RGB complex refractive index, imaginary part
float	roughness	0.1	roughness in [0–1], 0 is perfect mirror

The main appearance (mostly the color) of the Metal material is controlled by the physical parameters `eta` and `k`, the wavelength-dependent, complex index of refraction. These coefficients are quite counterintuitive but can be found in published measurements. For accuracy the index of refraction can be given as an array of spectral samples in `ior`, each sample a triplet of wavelength (in nm), eta, and k, ordered monotonically increasing by wavelength; OSPRay will then calculate the Fresnel in the spectral domain. Alternatively, `eta` and `k` can also be specified as approximated RGB coefficients; some examples are given in below table.

Metal	eta	k
Ag, Silver	(0.051, 0.043, 0.041)	(5.3, 3.6, 2.3)
Al, Aluminium	(1.5, 0.98, 0.6)	(7.6, 6.6, 5.4)
Au, Gold	(0.07, 0.37, 1.5)	(3.7, 2.3, 1.7)
Cr, Chromium	(3.2, 3.1, 2.3)	(3.3, 3.3, 3.1)
Cu, Copper	(0.1, 0.8, 1.1)	(3.5, 2.5, 2.4)

Table 3.28 – Index of refraction of selected metals as approximated RGB coefficients, based on data from https://refractiveindex.info/.

The `roughness` parameter controls the variation of microfacets and thus how polished the metal will look. The roughness can be modified by a texture `map_roughness` (texture transformations are supported as well) to create interesting edging effects.

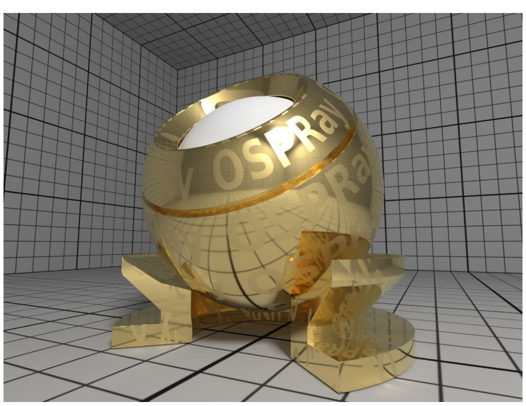

Figure 3.6 – Rendering of golden Metal material with textured roughness.

3.5.5.3 Alloy

The path tracer offers an alloy material, which behaves similar to Metal, but allows for more intuitive and flexible control of the color. To create an Alloy material pass the type string "`Alloy`" to `ospNewMaterial`. Its parameters are

Type	Name	Default	Description
vec3f	color	white 0.9	reflectivity at normal incidence (0 degree)
vec3f	edgeColor	white	reflectivity at grazing angle (90 degree)
float	roughness	0.1	roughness in [0–1], 0 is perfect mirror

Table 3.29 – Parameters of the Alloy material.

The main appearance of the Alloy material is controlled by the parameter `color`, while `edgeColor` influences the tint of reflections when seen at grazing angles (for real metals this is always 100% white). As in Metal the `roughness` parameter controls the variation of microfacets and thus how polished the alloy will look. All parameters can be textured by passing a texture handle, prefixed with "`map_`"; texture transformations are supported as well.

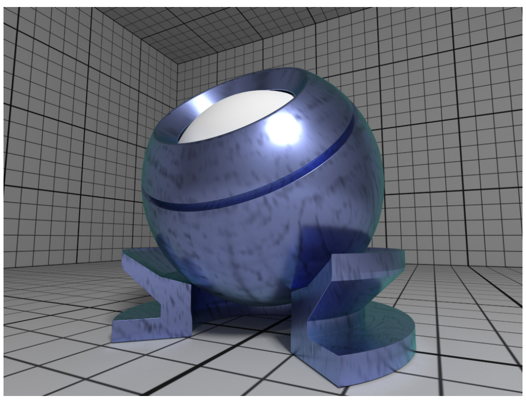

Figure 3.7 – Rendering of a fictional Alloy material with textured color.

3.5.5.4 Glass

The path tracer offers a realistic a glass material, supporting refraction and volumetric attenuation (i.e. the transparency color varies with the geometric thickness). To create a Glass material pass the type string "`Glass`" to `ospNewMaterial`. Its parameters are

Type	Name	Default	Description
float	eta	1.5	index of refraction
vec3f	attenuationColor	white	resulting color due to attenuation
float	attenuationDistance	1	distance affecting attenuation

Table 3.30 – Parameters of the Glass material.

For convenience, the rather counterintuitive physical attenuation coefficients will be calculated from the user inputs in such a way, that the `attenuationColor` will be the result when white light traveled trough a glass of thickness `attenuationDistance`.

3.5.5.5 ThinGlass

The path tracer offers a thin glass material useful for objects with just a single surface, most prominently windows. It models a very thin, transparent slab, i.e. it

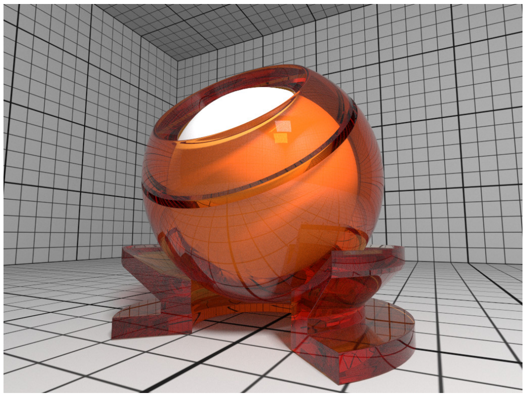

Figure 3.8 – Rendering of a Glass material with orange attenuation.

behaves as if a second, virtual surface is parallel to the real geometric surface. The implementation accounts for multiple internal reflections between the interfaces (including attenuation), but neglects parallax effects due to its (virtual) thickness. To create a such a thin glass material pass the type string "`ThinGlass`" to `ospNewMaterial`. Its parameters are

Type	Name	Default	Description
float	eta	1.5	index of refraction
vec3f	attenuationColor	white	resulting color due to attenuation
float	attenuationDistance	1	distance affecting attenuation
float	thickness	1	virtual thickness

Table 3.31 – Parameters of the ThinGlass material.

For convenience the attenuation is controlled the same way as with the Glass material. Additionally, the color due to attenuation can be modulated with a texture `map_attenuationColor` (texture transformations are supported as well). The `thickness` parameter sets the (virtual) thickness and allows for easy exchange of parameters with the (real) Glass material; internally just the ratio between `attenuationDistance` and `thickness` is used to calculate the resulting attenuation and thus the material appearance.

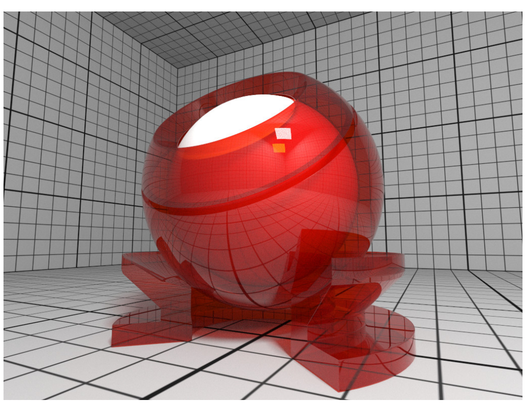

Figure 3.9 – Rendering of a ThinGlass material with red attenuation.

3.5.5.6 Luminous

The path tracer supports the Luminous material which emits light uniformly in all directions and which can thus be used to turn any geometric object into a light source. It is created by passing the type string "`Luminous`" to `ospNewMaterial`. The amount of constant radiance that is emitted is determined by combining the general parameters of lights: `color and intensity`.

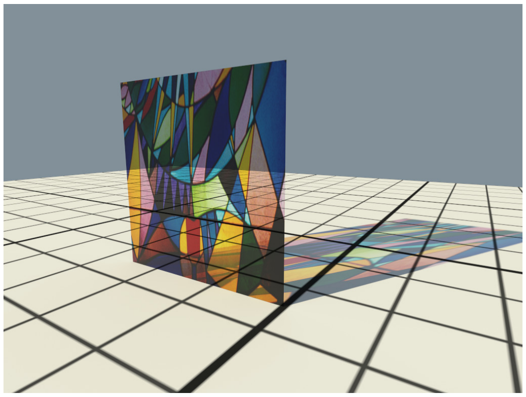

Figure 3.10 – Example image of a colored window made with textured attenuation of the ThinGlass material.

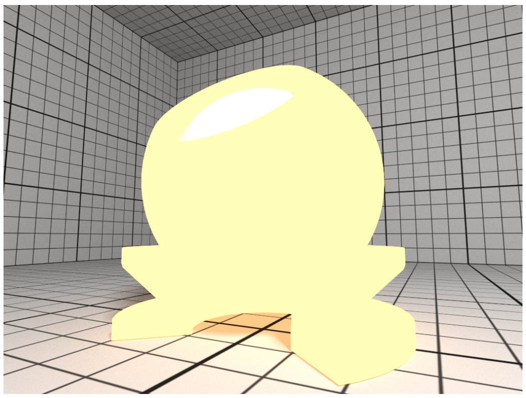

Figure 3.11 – Rendering of a yellow Luminous material.

3.5.6 Texture

To create a new 2D texture of size `size` (in pixels) and with the given format and flags use

```
OSPTexture2D ospNewTexture2D(const vec2i &size,
                             const OSPTextureFormat,
                             void *source = NULL,
                             const uint32_t textureCreationFlags = 0);
```

The call returns `NULL` if the texture could not be created with the given parameters, or else an `OSPTexture2D` handle to the created texture. The supported texture formats are:

The texel data addressed by `source` starts with the texels in the lower left corner of the texture image, like in OpenGL. Similar to data buffers the texel data can be shared by the application by specifying the `OSP_TEXTURE_SHARED_BUFFER` flag. Per default a texture fetch is filtered by performing bi-linear interpolation of the nearest 2×2 texels; if instead fetching only the nearest texel is desired (i.e. no filtering) then pass the `OSP_TEXTURE_FILTER_NEAREST` flag. Both texture creating flags can be combined with a bitwise OR.

3.5.7 Texture Transformations

All materials with textures also offer to manipulate the placement of these textures with the help of texture transformations. If so, this convention shall be used. The following parameters (prefixed with "`texture_name.`") are combined into one transformation matrix:

The transformations are applied in the given order. Rotation, scale and translation are interpreted "texture centric", i.e. their effect seen by an user are relative to the texture (although the transformations are applied to the texture coordinates).

Name	Description
OSP_TEXTURE_RGBA8	8 bit [0–255] linear components red, green, blue, alpha
OSP_TEXTURE_SRGBA	8 bit sRGB gamma encoded color components, and linear alpha
OSP_TEXTURE_RGBA32F	32 bit float components red, green, blue, alpha
OSP_TEXTURE_RGB8	8 bit [0–255] linear components red, green, blue
OSP_TEXTURE_SRGB	8 bit sRGB gamma encoded components red, green, blue
OSP_TEXTURE_RGB32F	32 bit float components red, green, blue
OSP_TEXTURE_R8	8 bit [0–255] linear single component
OSP_TEXTURE_R32F	32 bit float single component

Table 3.32 – Supported texture formats by `ospNewTexture2D`, i.e. valid constants of type `OSPTextureFormat`.

Type	Name	Description
vec4f	transform	interpreted as 2×2 matrix (linear part), column-major
float	rotation	angle in degree, counterclock-wise, around center
vec2f	scale	enlarge texture, relative to center (0.5, 0.5)
vec2f	translation	move texture in positive direction (right/up)

Table 3.33 – Parameters to define texture coordinate transformations.

3.5.8 Cameras

To create a new camera of given type `type` use

```
OSPCamera ospNewCamera(const char *type);
```

The call returns `NULL` if that type of camera is not known, or else an `OSPCamera` handle to the created camera. All cameras accept these parameters:

Type	Name	Description
vec3f(a)	pos	position of the camera in world-space
vec3f(a)	dir	main viewing direction of the camera
vec3f(a)	up	up direction of the camera
float	nearClip	near clipping distance
vec2f	imageStart	start of image region (lower left corner)
vec2f	imageEnd	end of image region (upper right corner)

Table 3.34 – Parameters accepted by all cameras.

The camera is placed and oriented in the world with `pos`, `dir` and `up`. OSPRay uses a right-handed coordinate system. The region of the camera sensor that is rendered to the image can be specified in normalized screen-space coordinates with `imageStart` (lower left corner) and `imageEnd` (upper right corner). This can be used, for example, to crop the image, to achieve asymmetrical view frusta, or to horizontally flip the image to view scenes which are specified in a left-handed coordinate system. Note that values outside the default range of [0–1] are valid, which is useful to easily realize overscan or film gate, or to emulate a shifted sensor.

3.5.8.1 Perspective Camera

The perspective camera implements a simple thinlens camera for perspective rendering, supporting optionally depth of field and stereo rendering, but no motion blur. It is created by passing the type string "`perspective`" to `ospNewCamera`. In addition to the general parameters understood by all cameras the perspective camera supports the special parameters listed in the table below.

Table 3.35 – Parameters accepted by the perspective camera.

Type	Name	Description
float	fovy	the field of view (angle in degree) of the frame's height
float	aspect	ratio of width by height of the frame
float	apertureRadius	size of the aperture, controls the depth of field
float	focusDistance	distance at where the image is sharpest when depth of field is enabled
bool	architectural	vertical edges are projected to be parallel
int	stereoMode	0: no stereo (default), 1: left eye, 2: right eye, 3: side-by-side
float	interpupillaryDistance	distance between left and right eye when stereo is enabled

Note that when setting the `aspect` ratio a non-default image region (using `imageStart` & `imageEnd`) needs to be regarded.

In architectural photography it is often desired for aesthetic reasons to display the vertical edges of buildings or walls vertically in the image as well, regardless of how the camera is tilted. Enabling the `architectural` mode achieves this by internally leveling the camera parallel to the ground (based on the `up` direction) and then shifting the lens such that the objects in direction `dir` are centered in the image. If finer control of the lens shift is needed use `imageStart` & `imageEnd`. Because the camera is now effectively leveled its image plane and thus the plane of focus is oriented parallel to the front of buildings, the whole façade appears sharp, as can be seen in the example images below.

3.5.8.2 Orthographic Camera

The orthographic camera implements a simple camera with orthographic projection, without support for depth of field or motion blur. It is created by passing the type string "`orthographic`" to `ospNewCamera`. In addition to the general parameters understood by all cameras the orthographic camera supports the following special parameters:

Type	Name	Description
float	height	size of the camera's image plane in y, in world coordinates
float	aspect	ratio of width by height of the frame

Table 3.36 – Parameters accepted by the orthographic camera.

For convenience the size of the camera sensor, and thus the extent of the scene that is captured in the image, can be controlled with the `height` parameter. The same effect can be achieved with `imageStart` and `imageEnd`, and both methods can be combined. In any case, the `aspect` ratio needs to be set accordingly to get an undistorted image.

3.5.8.3 Panoramic Camera

The panoramic camera implements a simple camera without support for motion blur. It captures the complete surrounding with a latitude / longitude mapping

Figure 3.12 – Example image created with the perspective camera, featuring depth of field.

Figure 3.13 – Enabling the `architectural` flag corrects the perspective projection distortion, resulting in parallel vertical edges.

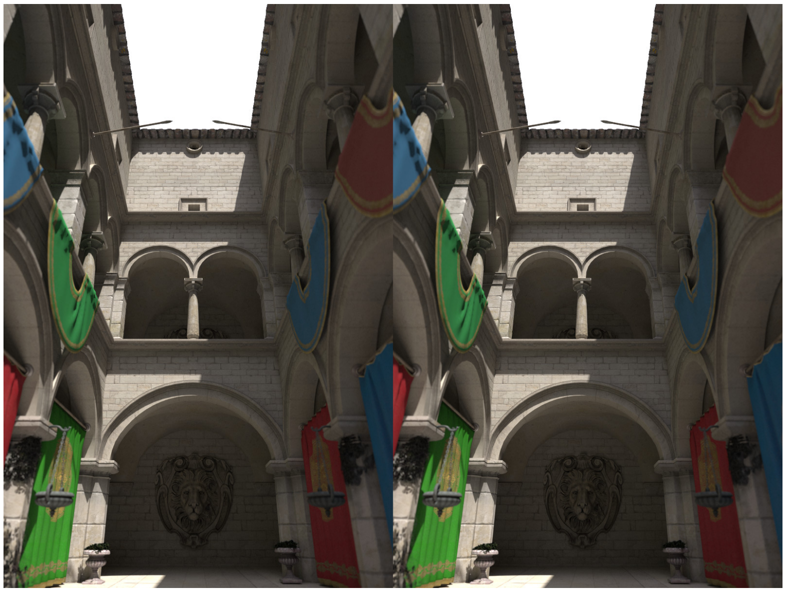

Figure 3.14 – Example 3D stereo image using `stereoMode` side-by-side.

Figure 3.15 – Example image created with the orthographic camera.

and thus the rendered images should best have a ratio of 2:1. A panoramic camera is created by passing the type string "`panoramic`" to `ospNewCamera`. It is placed and oriented in the scene by using the general parameters understood by all cameras.

Figure 3.16 – Latitude / longitude map created with the panoramic camera.

3.5.9 Picking

To get the world-space position of the geometry (if any) seen at [0–1] normalized screen-space pixel coordinates `screenPos` use

```
void ospPick(OSPPickResult*, OSPRenderer, const vec2f &screenPos);
```

The result is returned in the provided `OSPPickResult` struct:

```
typedef struct {
    vec3f position; // the position of the hit point (in world-space)
    bool hit;       // whether or not a hit actually occurred
} OSPPickResult;
```

Note that `ospPick` considers exactly the same camera of the given renderer that is used to render an image, thus matching results can be expected. If the camera supports depth of field then the center of the lens and thus the center of the circle of confusion is used for picking.

3.6 Framebuffer

The framebuffer holds the rendered 2D image (and optionally auxiliary information associated with pixels). To create a new framebuffer object of given size `size` (in pixels), color format, and channels use

```
OSPFrameBuffer ospNewFrameBuffer(const vec2i &size,
                                 const OSPFrameBufferFormat format = OSP_FB_SRGBA,
                                 const uint32_t frameBufferChannels = OSP_FB_COLOR);
```

The parameter `format` describes the format the color buffer has *on the host*, and the format that `ospMapFrameBuffer` will eventually return. Valid values are:

The parameter `frameBufferChannels` specifies which channels the framebuffer holds, and can be combined together by bitwise OR from the values of `OSPFrameBufferChannel` listed in the table below.

If a certain channel value is *not* specified, the given buffer channel will not be present. Note that ospray makes a very clear distinction between the *external* format of the framebuffer and the internal one: The external format is the

Name	Description
OSP_FB_NONE	framebuffer will not be mapped by the application
OSP_FB_RGBA8	8 bit [0–255] linear component red, green, blue, alpha
OSP_FB_SRGBA	8 bit sRGB gamma encoded color components, and linear alpha
OSP_FB_RGBA32F	32 bit float components red, green, blue, alpha

Table 3.37 – Supported color formats of the framebuffer that can be passed to `ospNewFrameBuffer`, i.e. valid constants of type `OSPFrameBufferFormat`.

Name	Description
OSP_FB_COLOR	RGB color including alpha
OSP_FB_DEPTH	euclidean distance to the camera (*not* to the image plane)
OSP_FB_ACCUM	accumulation buffer for progressive refinement
OSP_FB_VARIANCE	estimate of the current variance if OSP_FB_ACCUM is also present, see rendering

Table 3.38 – Framebuffer channels constants (of type `OSPFrameBufferChannel`), naming optional information the framebuffer can store. These values can be combined by bitwise OR when passed to `ospNewFrameBuffer` or `ospClearFrameBuffer`.

format the user specifies in the `format` parameter; it specifies what color format OSPRay will eventually *return* the framebuffer to the application (when calling `ospMapFrameBuffer`): no matter what OSPRay uses internally, it will simply return a 2D array of pixels of that format, with possibly all kinds of reformatting, compression/decompression, etc, going on in-between the generation of the *internal* framebuffer and the mapping of the externally visible one.

In particular, `OSP_FB_NONE` is a perfectly valid pixel format for a framebuffer that an application will never map. For example, an application driving a display wall may well generate an intermediate framebuffer and eventually transfer its pixel to the individual displays using an `OSPPixelOp` pixel operation.

A framebuffer can be freed again using

```
void ospFreeFrameBuffer(OSPFrameBuffer);
```

Because OSPRay uses reference counting internally the framebuffer may not immediately be deleted at this time.

The application can map the given channel of a framebuffer – and thus access the stored pixel information – via

```
const void *ospMapFrameBuffer(OSPFrameBuffer,
                              const OSPFrameBufferChannel = OSP_FB_COLOR);
```

Note that only `OSP_FB_COLOR` or `OSP_FB_DEPTH` can be mapped. The origin of the screen coordinate system in OSPRay is the lower left corner (as in OpenGL), thus the first pixel addressed by the returned pointer is the lower left pixel of the image.

A previously mapped channel of a framebuffer can be unmapped by passing the received pointer `mapped` to

```
void ospUnmapFrameBuffer(const void *mapped, OSPFrameBuffer);
```

The individual channels of a framebuffer can be cleared with

```
void ospFrameBufferClear(OSPFrameBuffer, const uint32_t frameBufferChannels);
```

When selected, `OSP_FB_COLOR` will clear the color buffer to black `(0, 0, 0, 0)`, `OSP_FB_DEPTH` will clear the depth buffer to `inf`, `OSP_FB_ACCUM` will clear the accumulation buffer to black, resets the accumulation counter `accumID` and also clears the variance buffer (if present) to `inf`.

Pixel Operation

A pixel operation are functions that are applied to every pixel that gets written into a framebuffer. Examples include post-processing, filtering, blending, tone mapping, or sending tiles to a display wall. To create a new pixel operation of given type `type` use

```
OSPPixelOp ospNewPixelOp(const char *type);
```

The call returns `NULL` if that type is not known, or else an `OSPPixelOp` handle to the created pixel operation.

To set a pixel operation to the given framebuffer use

```
void ospSetPixelOp(OSPFrameBuffer, OSPPixelOp);
```

3.7 Rendering

To render a frame into the given framebuffer with the given renderer use

```
float ospRenderFrame(OSPFrameBuffer, OSPRenderer,
                     const uint32_t frameBufferChannels = OSP_FB_COLOR);
```

The third parameter specifies what channel(s) of the framebuffer is written to[4]. What to render and how to render it depends on the renderer's parameters. If the framebuffer supports accumulation (i.e. it was created with `OSP_FB_ACCUM`) then successive calls to `ospRenderFrame` will progressively refine the rendered image. If additionally the framebuffer has an `OSP_FB_VARIANCE` channel then `ospRenderFrame` returns an estimate of the current variance of the rendered image, otherwise `inf` is returned. The estimated variance can be used by the application as a quality indicator and thus to decide whether to stop or to continue progressive rendering.

[4] This is currently not implemented, i.e. all channels of the framebuffer are always updated.

Chapter 4

Parallel Rendering with MPI

OSPRay has the ability to scale to multiple nodes in a cluster via MPI. This enables applications to take advantage of larger compute and memory resources when available.

4.1 Prerequisites for MPI Mode

In addition to the standard build requirements of OSPRay, you must have the following items available in your environment in order to build&run OSPRay in MPI mode:

- An MPI enabled multi-node environment, such as an HPC cluster
- An MPI implementation you can build against (i.e. Intel MPI, MVAPICH2, etc…)

4.2 Enabling the MPI Module in your Build

To build the MPI module the CMake option `OSPRAY_MODULE_MPI` must be enabled, which can be done directly on the command line (with `-DOSPRAY_MODULE_MPI=ON`) or through a configuration dialog (`ccmake`, `cmake-gui`), see also Compiling OSPRay.

This will trigger CMake to go look for an MPI implementation in your environment. You can then inspect the CMake value of `MPI_LIBRARY` to make sure that CMake found your MPI build environment correctly.

This will result in an OSPRay module being built. To enable using it, applications will need to either link `libospray_module_mpi`, or call

```
ospLoadModule("mpi");
```

before initializing OSPRay.

4.3 Modes of Using OSPRay's MPI Features

OSPRay provides two ways of using MPI to scale up rendering: offload and distributed.

4.3.1 Offload Rendering

The "offload" rendering mode is where a single (not-distributed) calling application treats the OSPRay API the same as with local rendering. However, OSPRay uses multiple MPI connected nodes to evenly distribute frame rendering work, where each node contains a full copy of all scene data. This method is most effective for scenes which can fit into memory, but are very expensive to render: for example, path tracing with many samples-per-pixel is very compute heavy, making it a good situation to use the offload feature. This can be done with any application which already uses OSPRay for local rendering without the need for any code changes.

When doing MPI offload rendering, applications can optionally enable dynamic load balancing, which can be beneficial in certain contexts. This load balancing refers to the distribution of tile rendering work across nodes: thread-level load balancing on each node is still dynamic with the thread tasking system. The options for enabling/controlling the dynamic load balacing features on the `mpi_offload` device are found in the table below, which can be changed while the application is running. Please note that these options will likely only pay off for scenes which have heavy rendering load (e.g. path tracing a non-trivial scene) and have a lot of variance in how expensive each tile is to render.

Type	Name	Default	Description
bool	dynamicLoadBalancer	false	whether to use dynamic load balancing

Table 4.1 – Parameters specific to the `mpi_offload` device

4.3.2 Distributed Rendering

The "distributed" rendering mode is where a MPI distributed application (such as a scientific simulation) uses OSPRay collectively to render frames. In this case, the API expects all calls (both created objects and parameters) to be the same on every application rank, except each rank can specify arbitrary geometries and volumes. Each renderer will have its own limitations on the topology of the data (i.e. overlapping data regions, concave data, etc.), but the API calls will only differ for scene objects. Thus all other calls (i.e. setting camera, creating framebuffer, rendering frame, etc.) will all be assumed to be identical, but only rendering a frame and committing the model must be in lock-step. This mode targets using all available aggregate memory for very large scenes and for "in-situ" visualization where the data is already distributed by a simulation app.

4.4 Running an Application with the "offload" Device

As an example, our sample viewer can be run as a single application which offloads rendering work to multiple MPI processes running on multiple machines.

The example apps are setup to be launched in two different setups. In either setup, the application must initialize OSPRay with the offload device. This can be done by creating an "`mpi_offload`" device and setting it as the current device (via the `ospSetCurrentDevice()` function), or passing either "`--osp:mpi`" or "`--osp:mpi-offload`" as a command line parameter to `ospInit()`. Note that passing a command line parameter will automatically call `ospLoadModule("mpi")` to load the MPI module, while the application will have to load the module explicitly if using `ospNewDevice()`.

4.4.1 Single MPI Launch

OSPRay is initialized with the `ospInit()` function call which takes command line arguments in and configures OSPRay based on what it finds. In this setup, the app is launched across all ranks, but workers will never return from `ospInit()`, essentially turning the application into a worker process for OSPRay. Here's an example of running the ospVolumeViewer data-replicated, using `c1-c4` as compute nodes and `localhost` the process running the viewer itself:

```
mpirun -perhost 1 -hosts localhost,c1,c2,c3,c4 ./ospExampleViewer <scene file> --osp:mpi
```

4.4.2 Separate Application&Worker Launches

The second option is to explicitly launch the app on rank 0 and worker ranks on the other nodes. This is done by running `ospray_mpi_worker` on worker nodes and the application on the display node. Here's the same example above using this syntax:

```
mpirun -perhost 1 -hosts localhost ./ospExampleViewer <scene file> --osp:mpi \
  : -hosts c1,c2,c3,c4 ./ospray_mpi_worker
```

This method of launching the application and OSPRay worker separately works best for applications which do not immediately call `ospInit()` in their `main()` function, or for environments where application dependencies (such as GUI libraries) may not be available on compute nodes.

4.5 Running an Application with the "distributed" Device

Applications using the new distributed device should initialize OSPRay by creating (and setting current) an "`mpi_distributed`" device or pass `"--osp:mpi-distributed"` as a command line argument to `ospInit()`. Note that due to the semantic differences the distributed device gives the OSPRay API, it is not expected for applications which can already use the offload device to correctly use the distributed device without changes to the application.

Chapter 5

Examples

5.1 Tutorial

A minimal working example demonstrating how to use OSPRay can be found at `apps/ospTutorial.c`[1]. On Linux build it in the build directory with

```
gcc -std=c99 ../apps/ospTutorial.c -I ../ospray/include -I .. \
./libospray.so -Wl,-rpath,. -o ospTutorial
```

[1] A C++ version that uses the C++ convenience wrappers of OSPRay's C99 API via `include/ospray/ospray_cpp.h` is available at `apps/ospTutorial.cpp`.

On Windows build it in the "build_directory\$Configuration" with

```
cl ..\..\apps\ospTutorial.c -I ..\..\ospray\include -I ..\.. ospray.lib
```

Running `ospTutorial` will create two images of two triangles, rendered with the Scientific Visualization renderer with full Ambient Occlusion. The first image `firstFrame.ppm` shows the result after one call to `ospRenderFrame` – jagged edges and noise in the shadow can be seen. Calling `ospRenderFrame` multiple times enables progressive refinement, resulting in antialiased edges and converged shadows, shown after ten frames in the second image `accumulatedFrames.ppm`.

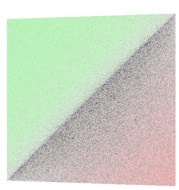

Figure 5.1 – First frame.

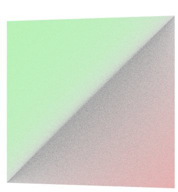

Figure 5.2 – After accumulating ten frames.

5.2 Example Viewer

OSPRay also includes an exemplary viewer application `ospExampleViewerSg`, showcasing all features of OSPRay. The Example Viewer uses the ImGui library for user interface controls. The viewer is based on a prototype OSPRay scenegraph interface where its nodes are displayed in the GUI and can be manipulated interactively. For instance, simply run it as `ospExampleViewerSg teapot.obj`.

This application also functions as an OSPRay state debugger – invalid values will be shown in red up the hierarchy and won't change the viewer until corrected. You can also add new nodes where appropriate: for example, when "lights" is expanded right clicking on "lights" and typing in a light type, such as "point", will add it to the scene. Similarly, right clicking on "world" and creating an "Importer" node will add a new scene importer from a file. Changing the filename to an appropriate file will load the scene and propagate the resulting state.

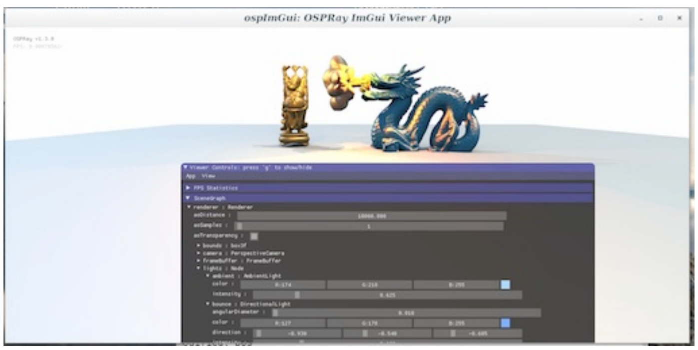

Figure 5.3 – Screenshot of `ospExampleViewerSg`

5.3 Distributed Viewer

The application `ospDistribViewerDemo` demonstrates how to write a distributed SciVis style interactive renderer using the distributed MPI device. Note that because OSPRay uses sort-last compositing it is up to the user to ensure that the data distribution across the nodes is suitable. Specifically, each nodes' data must be convex and disjoint. This renderer supports multiple volumes and geometries per node. To ensure they are composited correctly you specify a list of bounding regions to the model, within these regions can be arbitrary volumes/geometries and each rank can have as many regions as needed. As long as the regions are disjoint/convex the data will be rendered correctly. In this demo we either generate a volume, or load a RAW volume file if one is passed on the commandline.

5.3.1 Loading a RAW Volume

To load a RAW volume you must specify the filename (`-f <file>`), the data type (`-dtype <dtype>`), the dimensions (`-dims <x> <y> <z>`) and the value range for the transfer function (`-range <min> <max>`). For example, to run on the CSAFE dataset from the demos page you would pass the following arguments:

```
mpirun -np <n> ./ospDistribViewerDemo \
    -f <path to csafe>/csafe-heptane-302-volume.raw \
    -dtype uchar -dims 302 302 302 -range 0 255
```

The volume file will then be chunked up into an `x×y×z` grid such that $n = xyz$. See `loadVolume` in gensv/generateSciVis.cpp for an example of how to properly load a volume distributed across ranks with correct specification of brick positions and ghost voxels for interpolation at boundaries. If no volume file data is passed a volume will be generated instead, in that case see `makeVolume`.

5.3.2 Geometry

The viewer can also display some randomly generated sphere geometry if you pass `-spheres <n>` where `n` is the number of spheres to generate per-node. These spheres will be generated inside the bounding box of the region's volume data.

In the case that you have geometry crossing the boundary of nodes and are replicating it on both nodes to render (ghost zones, etc.) the region will be used by the renderer to clip rays against allowing to split the object between the two nodes, with each rendering half. This will keep the regions rendered by each rank disjoint and thus avoid any artifacts. For example, if a sphere center is on the border between two nodes, each would render half the sphere and the halves would be composited to produce the final complete sphere in the image.

5.3.3 App-initialized MPI

Passing the `-appMPI` flag will have the application initialize MPI instead of letting OSPRay do it internally when creating the MPI distributed device. In this case OSPRay will not finalize MPI when cleaning up the device, allowing the application to use OSPRay for some work, shut it down and recreate everything later if needed for additional computation, without accidentally shutting down its MPI communication.

5.3.4 Interactive Viewer

Rank 0 will open an interactive window with GLFW and display the rendered image. When the application state needs to update (e.g. camera or transfer function changes), this information is broadcasted out to the other nodes to update their scene data.

5.4 Demos

Several ready-to-run demos, models and data sets for OSPRay can be found at the OSPRay Demos and Examples page.