Contents

1 OSPRay Overview 4
  1.1 OSPRay Support and Contact 4
  1.2 Version History 4

2 Building and Finding OSPRay 19
  2.1 Prerequisites 19
  2.2 CMake Superbuild 20
  2.3 Standard CMake build 21
  2.4 Finding an OSPRay Install with CMake 22

3 OSPRay API 23
  3.1 Initialization and Shutdown 23
    3.1.1 Command Line Arguments 23
    3.1.2 Manual Device Instantiation 23
    3.1.3 Environment Variables 25
    3.1.4 Error Handling and Status Messages 25
    3.1.5 Loading OSPRay Extensions at Runtime 27
    3.1.6 Shutting Down OSPRay 27
  3.2 Objects 27
    3.2.1 Parameters 28
    3.2.2 Data 28
  3.3 Volumes 30
    3.3.1 Structured Regular Volume 30
    3.3.2 Structured Spherical Volume 31
    3.3.3 Adaptive Mesh Refinement (AMR) Volume 32
    3.3.4 Unstructured Volume 33
    3.3.5 Transfer Function 33
    3.3.6 VolumetricModels 33
  3.4 Geometries 34
    3.4.1 Mesh 35
    3.4.2 Subdivision 35
    3.4.3 Spheres 35
    3.4.4 Curves 35
    3.4.5 Boxes 38
    3.4.6 Isosurfaces 38
    3.4.7 GeometricModels 38
  3.5 Lights 38
    3.5.1 Directional Light / Distant Light 39
    3.5.2 Point Light / Sphere Light 39
    3.5.3 Spotlight 40
    3.5.4 Quad Light 40
    3.5.5 HDRI Light 41
    3.5.6 Ambient Light 41
    3.5.7 Emissive Objects 41
3.6 Scene Hierarchy ........................................... 41
  3.6.1 Groups .................................................. 41
  3.6.2 Instances ............................................... 42
  3.6.3 World ................................................ 42
3.7 Renderers .................................................. 43
  3.7.1 SciVis Renderer ......................................... 44
  3.7.2 Path Tracer ............................................ 44
  3.7.3 Materials .............................................. 44
  3.7.4 Texture ............................................... 53
  3.7.5 Texture2D Transformations .......................... 54
  3.7.6 Cameras .............................................. 54
  3.7.7 Picking .............................................. 57
3.8 Framebuffer ............................................... 59
  3.8.1 Image Operation ...................................... 60
3.9 Rendering .................................................. 61
  3.9.1 Asynchronous Rendering .............................. 61
  3.9.2 Asynchronously Rendering and ospCommit() ...... 62
  3.9.3 Synchronous Rendering ............................... 62
3.10 Distributed rendering with MPI .......................... 63

4 Examples .................................................... 64
  4.1 Tutorial ................................................ 64
  4.2 ospExamples ............................................ 65
Chapter 1

OSPRay Overview

Intel OSPRay is an open source, scalable, and portable ray tracing engine for high-performance, high-fidelity visualization on Intel Architecture CPUs. OSPRay is part of the Intel oneAPI Rendering Toolkit and 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 Intel Embree and ISPC (Intel SPMD Program Compiler), and fully exploits 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.

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

1.2 Version History

1.2.1 Changes in v2.0.1:

- Fix bug where Embree user-defined geometries were not indexed correctly in the scene, which now requires Embree v3.8.0+
- Fix crash when the path tracer encounters geometric models that do not have a material
- Fix crash when some path tracer materials generated NULL bsdfs
- Fix bug where ospGetBounds returned incorrect values
- Fix missing symbol in denoiser module
- Fix missing symbol exports on Windows for all OSPRay built modules
- Add the option to specify a single color for geometric models
- The scivis renderer now respects the opacity component of color on geometric models
• Fix various inconsistent handling of frame buffer alpha between renderers
• ospGetCurrentDevice now increments the ref count of the returned OSPDevice handle, so applications will need to release the handle when finished by using ospDeviceRelease accordingly
• Added denoiser to ospExamples app
• Added module_mpi to superbuild (disabled by default)
• The superbuild now will emit a CMake error when using any 32-bit CMake generator, as 32-bit builds are not supported

1.2.2 Changes in v2.0.0:

• New major revision of OSPRay brings API breaking improvements over v1.x. See doc/ospray2_porting_guide.md for a deeper description of migrating from v1.x to v2.0 and the latest API documentation
  – ospRenderFrame now takes all participating objects as function parameters instead of setting some as renderer params
  – ospRenderFrame is now asynchronous, where the task is managed through a returned OSPFuture handle
  – The heirarchy of objects in a scene are now more granular to aid in scene construction flexibility and reduce potential object duplication
  – Type-specific parameter setting functions have been consolidated into a single ospSetParam API call
  – C++ wrappers found in ospray_cpp.h now automatically track handle lifetimes, therefore applications using them do not need to use ospRelease (or the new ospRetain) with them: see usage example in apps/tutorials/ospTutorial.cpp
  – Unused parameters are reported as status messages when logLevel is at least warning (most easily set by enabling OSPRay debug on initialization)

• New utility library which adds functions to help with new API migration and reduction of boilerplate code
  – Use ospray_util.h to access these additional functions
  – All utility functions are implemented in terms of the core API found in ospray.h, therefore they are compatible with any device backend

• Introduction of new Intel® Open Volume Kernel Library (Open VKL) for greatly enhanced volume sampling and rendering features and performance
• Added direct support for Intel® Open Image Denoise as an optional module, which adds a denoiser type to ospNewImageOperation
• OSPRay now requires minimum Embree v3.7.0
• New CMake superbuild available to build both OSPRay’s dependencies and OSPRay itself
  – Found in scripts/superbuild
  – See documentation for more details and example usage
• The ospcommon library now lives as a stand alone repository and is required to build OSPRay
• The MPI module is now a separate repository, which also contains all MPI distributed rendering documentation
• Log levels are now controled with enums and named strings (where applicable)
  – A new flag was also introduced which turns all OSP_LOG_WARNING messages into errors, which are submitted to the error callback instead of the message callback
Any unused parameters an object ignores now emit a warning message

- New support for volumes in the pathtracer
  - Several parameters are available for performance/quality trade-offs for both photo-realistic and scientific visualization use cases

- Simplification of the SciVis renderer
  - Fixed AO lighting and simple ray marched volume rendering for ease of use and performance

- Overlapping volumes are now supported in both the pathtracer and scivis renderers
- New API call for querying the bounds of objects (OSPWorld, OSPInstance, and OSPGroup)
- Lights now exist as a parameter to the world instead of the renderer
- Removal of slices geometry. Instead, any geometry with volume texture can be used for slicing
- Introduction of new boxes geometry type
- Expansion of information returned by ospPick
- Addition of API to query version information at runtime
- Curves now supports both, per vertex varying radii as in vec4f[] vertex.position_radius and constant radius for the geometry with float radius. It uses OSP_ROUND type and OSP_LINEAR basis by default to create the connected segments of constant radius. For per vertex varying radii curves it uses Embree curves.
  - Add new Embree curve type OSP_CATMULL_ROM for curves
  - Minimum required Embree version is now 3.7.0

- Removal of cylinders and streamlines geometry, use curves instead
- Triangle mesh and Quad mesh are superseded by the mesh geometry
- Applications need to use the various error reporting methods to check whether the creation (via ospNew...) of objects failed; a returned NULL is not a special handle anymore to signify an error
- Changed module init methods to facilitate version checking: extern "C" OSPError ospray_module_init_<name>(int16_t versionMajor, int16_t versionMinor, int16_t versionPatch)
- The map_backplate texture is supported in all renderers and does not hide lights in infinity (like the HDRI light) anymore; explicitly make lights invisible if this is needed
- Changed the computation of variance for adaptive accumulation to be independent of TILE_SIZE, thus varianceThreshold needs to be adapted if using a different TILE_SIZE than default 64
- OSPGeometricModel now has the option to index a renderer-global material list that lives on the renderer, allowing scenes to avoid rendererspecific materials
- Object type names and parameters all now follow the camel-case convention
- New ospExamples app which consolidates previous interactive apps into one
- New ospBenchmark app which implements a runnable benchmark suite

Known issues:
- ISPC v1.11.0 and Embree v3.6.0 are both incompatible with OSPRay and should be avoided (OSPRay should catch this during CMake configure)
1.2.3 Changes in v1.8.5:
- Fix float precision corner case (NaNs) in sphere light sampling
- Fix CMake bug that assumed .git was a directory, which is not true when using OSPRay as a git submodule
- Fix CMake warning
- Fix DLL_EXPORT issue with ospray_testing helper library on Windows

1.2.4 Changes in v1.8.4:
- Add location of libospray to paths searched to find modules

1.2.5 Changes in v1.8.3:
- Fix bug where parameters set by ospSet1b() were being ignored
- Fix bug in box intersection tests possibly creating NaNs
- Fix issue with client applications calling find_package(ospray) more than once
- Fix bug in cylinder intersection when ray and cylinder are perpendicular
- Fix rare crash in path tracer / MultiBSDF

1.2.6 Changes in v1.8.2:
- CMake bug fix where external users of OSPRay needed CMake newer than version 3.1
- Fix incorrect propagation of tasking system flags from an OSPRay install
- Fix inconsistency between supported environment variables and command line parameters passed to ospInit()
  - Missing variables were OSPRAY_LOAD_MODULES and OSPRAY_DEFAULT_DEVICE

1.2.7 Changes in v1.8.1:
- CMake bug fix to remove full paths to dependencies in packages

1.2.8 Changes in v1.8.0:
- This will be the last minor revision of OSPRay. Future development effort in the devel branch will be dedicated to v2.0 API changes and may break existing v1.x applications.
  - This will also be the last version of OSPRay to include ospray_sg and the Example Viewer. Users which depend on these should instead use the separate OSPRay Studio project, where ospray_sg will be migrated.
  - We will continue to support patch releases of v1.8.x in case of any reported bugs
- Refactored CMake to use newer CMake concepts
  - All targets are now exported in OSPRay installs and can be consumed by client projects with associated includes, libraries, and definitions
  - OSPRay now requires CMake v3.1 to build
  - See documentation for more details
- Added new minimal tutorial apps to replace the more complex Example Viewer
- Added new “subdivision” geometry type to support subdivision surfaces
• Added support for texture formats L8, LA8 (gamma-encoded luminance), and RA8 (linear two component). Note that the enum OSP_TEXTURE_FORMAT_INVALID changed its value, thus recompilation may be necessary.
• Automatic epsilon handling, which removes the “epsilon” parameter on all renderers
• Normals in framebuffer channel OSP_FB_NORMAL are now in world-space
• Added support for Intel® Open Image Denoise to the Example Viewer
  – This same integration will soon be ported to OSPRay Studio
• Fixed artifacts for scaled instances of spheres, cylinders and streamlines
• Improvements to precision of intersections with cylinders and streamlines
• Fixed Quadlight: the emitting side is now indeed in direction edge1 × edge2

1.2.9 Changes in v1.7.3:
• Make sure a “default” device can always be created
• Fixed ospNewTexture2D (completely implementing old behavior)
• Cleanup any shared object handles from the OS created from ospLoadModule()

1.2.10 Changes in v1.7.2:
• Fixed issue in mpi_offload device where ospRelease would sometimes not correctly free objects
• Fixed issue in ospray_sg where structured volumes would not properly release the underlying OSPRay object handles

1.2.11 Changes in v1.7.1:
• Fixed issue where the Principled material would sometimes show up incorrectly as black
• Fixed issue where some headers were missing from install packages

1.2.12 Changes in v1.7.0:
• Generalized texture interface to support more than classic 2D image textures, thus OSPTexture2D and ospNewTexture2D are now deprecated, use the new API call ospNewTexture(“texture2d”) instead
  – Added new volume texture type to visualize volume data on arbitrary geometry placed inside the volume
• Added new framebuffer channels OSP_FB_NORMAL and OSP_FB_ALBEDO
• Applications can get information about the progress of rendering the current frame, and optionally cancel it, by registering a callback function via ospSetProgressFunc()
• Lights are not tied to the renderer type, so a new function ospNewLight3() was introduced to implement this. Please convert all uses of ospNewLight() and/or ospNewLight2() to ospNewLight3()
• Added sheenTint parameter to Principled material
• Added baseNormal parameter to Principled material
• Added low-discrepancy sampling to path tracer
• The spp parameter on the renderer no longer supports values less than 1, instead applications should render to a separate, lower resolution framebuffer during interaction to achieve the same behavior
1.2.13 Changes in v1.6.1:

- Many bug fixes
  - Quad mesh interpolation and sampling
  - Normal mapping in path tracer materials
  - Memory corruption with partly emitting mesh lights
  - Logic for setting thread affinity

1.2.14 Changes in v1.6.0:

- Updated ispc device to use Embree3 (minimum required Embree version is 3.1)
- Added new `ospShutdown()` API function to aid in correctness and determinism of OSPRay API cleanup
- Added "Principled" and "CarPaint" materials to the path tracer
- Improved flexibility of the tone mapper
- Improvements to unstructured volume
  - Support for wedges (in addition to tets and hexes)
  - Support for implicit isosurface geometry
  - Support for cell-centered data (as an alternative to per-vertex data)
  - Added an option to precompute normals, providing a memory/performance trade-off for applications
- Implemented "quads" geometry type to handle quads directly
- Implemented the ability to set "void" cell values in all volume types: when NaN is present as a volume's cell value the volume sample will be ignored by the SciVis renderer
- Fixed support for color strides which were not multiples of `sizeof(float)`
- Added support for RGBA8 color format to spheres, which can be set by specifying the "color_format" parameter as `OSP_UCHAR4`, or passing the "color" array through an `OSPData` of type `OSP_UCHAR4`.
- Added ability to configure Embree scene flags via `OSPModel` parameters
- `ospFreeFrameBuffer()` has been deprecated in favor of using `ospRelease()` to free framebuffer handles
- Fixed memory leak caused by incorrect parameter reference counts in ispc device
- Fixed occasional crashes in the `mpi_offload` device on shutdown
- Various improvements to sample apps and `ospray_sg`
  - Added new generator nodes, allowing the ability to inject programmatically generated scene data (only C++ for now)
  - Bug fixes and improvements to enhance stability and usability

1.2.15 Changes in v1.5.0:

- Unstructured tetrahedral volume now generalized to also support hexahedral data, now called `unstructured_volume`
- New function for creating materials (`ospNewMaterial2()`) which takes the renderer type string, not a renderer instance (the old version is now deprecated)
- New `tonemapper PixelOp` for tone mapping final frames
- Streamlines now support per-vertex radii and smooth interpolation
- `ospray_sg` headers are now installed alongside the SDK
- Core OSPRay ispc device now implemented as a module
  - Devices which implement the public API are no longer required to link the dependencies to core OSPRay (e.g., Embree v2.x)
By default, \texttt{ospInit()} will load the ispc module if a device was not created via \texttt{--osp:mpi} or \texttt{--osp:device:[:name]}

- MPI devices can now accept an existing world communicator instead of always creating their own
- Added ability to control ISPC specific optimization flags via CMake options. See the various \texttt{ISPC_FLAGS_*} variables to control which flags get used
- Enhancements to sample applications
  - \texttt{ospray_sg} (and thus \texttt{ospExampleViewer/ospBenchmark}) can now be extended with new scene data importers via modules or the SDK
  - Updated \texttt{ospTutorial} examples to properly call \texttt{ospRelease()}
  - New options in the \texttt{ospExampleViewer} GUI to showcase new features (sRGB framebuffers, tone mapping, etc.)

- General bug fixes
  - Fixes to geometries with multiple emissive materials
  - Improvements to precision of ray-sphere intersections

1.2.16 Changes in v1.4.3:
- Several bug fixes
  - Fixed potential issue with static initialization order
  - Correct compiler flags for Debug config
  - Spheres \texttt{postIntersect} shading is now 64-bit safer

1.2.17 Changes in v1.4.2:
- Several cleanups and bug fixes
  - Fixed memory leak where the Embree BVH was never released when an \texttt{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 initialized when \texttt{spp <= 0}
  - Fixed an issue in \texttt{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

1.2.18 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
Additionally support RGB etal/k for Metal
- Added Alloy material, a “metal” with textured color
- Minimum required Embree version is now v2.15

1.2.19 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 scene graph implementation
  - Old (unused) libraries have been removed: miniSG, command line, 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

1.2.20 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

1.2.21 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 command line 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 do not 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

1.2.22 Changes in v1.2.1:
- Various bug fixes related to MPI distributed rendering, ISPC issues on Windows, and other build related issues

1.2.23 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 considerably 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

1.2.24 Changes in v1.1.2:
• Various bug fixes related to normalization, epsilons and debug messages

1.2.25 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

1.2.26 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

### 1.2.27 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 command line
  - 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

### 1.2.28 Changes in v0.10.1:

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

1.2.30 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

1.2.31 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 1.5×
• 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

1.2.32 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

1.2.33 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

1.2.34 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

1.2.35 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 (sub-)regions 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 runtime
• 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

1.2.36 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 and Finding OSPRay

The latest OSPRay sources are always available at the OSPRay GitHub repository. The default master branch should always point to the latest 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, MSVC, and Intel® C++ Compiler (icc)), and standard Linux development tools. To build the interactive tutorials, you should also have some version of OpenGL and GLFW.

- 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. Note: OSPRay is incompatible with ISPC v1.11.0.

- OSPRay builds on top of a small C++ utility library called ospcommon. The library provides abstractions for tasking, aligned memory allocation, vector math types, among others. For users who also need to build ospcommon, we recommend the default the Intel® Threading Building Blocks (TBB) as tasking system for performance and flexibility reasons. Alternatively you can set CMake variable OSPCOMMON_TASKING_SYSTEM to OpenMP or Internal.

- OSPRay also heavily uses Intel Embree, installing version 3.8.0 or newer is required. If Embree is not found by CMake its location can be hinted with the variable embree_DIR.

- OSPRay also heavily uses Intel Open VKL, installing version 0.8.0 or newer is required. If Open VKL is not found by CMake its location can be hinted with the variable openvkl_DIR.

- OSPRay also provides an optional module that adds support for Intel Open Image Denoise, which is enabled by OSPRAY_MODULE_DENOISER. When loaded, this module enables the denosier image operation. You may need

\(^1\) For example, if OSPRay is in ~/Projects/ospray, ISPC will also be searched in ~/Projects/ispc-v1.12.0-linux
Building and Finding OSPRay

2.1 Installing Dependencies

To hint the location of the library with the CMake variable OpenImageDenoise_DIR.

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:

```bash
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:

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

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

```bash
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 CMake Superbuild

For convenience, OSPRay provides a CMake Superbuild script which will pull down OSPRay’s dependencies and build OSPRay itself. By default, the result is an install directory, with each dependency in its own directory.

Run with:

```bash
mkdir build
cd build
cmake [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```

On Windows make sure to select the non-default 64bit generator, e.g.

```bash
cmake -G "Visual Studio 15 2017 Win64" [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```

The resulting install directory (or the one set with CMAKE_INSTALL_PREFIX) will have everything in it, with one subdirectory per dependency.

CMake options to note (all have sensible defaults):

- **CMAKE_INSTALL_PREFIX** will be the root directory where everything gets installed.
- **BUILD_JOBS** sets the number given to make -j for parallel builds.
- **INSTALL_IN_SEPARATE_DIRECTORIES** toggles installation of all libraries in separate or the same directory.
- **BUILD_EMBREE_FROM_SOURCE** set to OFF will download a pre-built version of Embree.
- **BUILD_OIDN_FROM_SOURCE** set to OFF will download a pre-built version of OpenImageDenoise.
- **BUILD_OIDN_VERSION** determines which version of OpenImageDenoise to pull down.

For the full set of options, run:

```bash
ccmake [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```

or

```bash
cmake-gui [<OSPRAY_SOURCE_LOC>/scripts/superbuild]
```
2.3 Standard CMake build

2.3.1 Compiling OSPRay on Linux and Mac OS X

Assuming 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

```
cmake ..
```

- Make sure to properly set build mode and enable the components you need, etc.; then type `configure` and `generate`. When back on the command prompt, build it using

```
make
```

- You should now have `libospray.[so,dylib]` as well as a set of example applications.

2.3.2 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 (OSPRay needs Visual Studio 14 2015 or newer), 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 and `openvkl_DIR` to where Open VKL 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 15 2017 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
```

### 2.4 Finding an OSPRay Install with CMake

Client applications using OSPRay can find it with CMake’s find_package() command. For example,

```
fnd_package(ospray 2.0.0 REQUIRED)
```

finds OSPRay via OSPRay’s configuration file osprayConfig.cmake. Once found, the following is all that is required to use OSPRay:

```
target_link_libraries(${client_target} ospray::ospray)
```

This will automatically propagate all required include paths, linked libraries, and compiler definitions to the client CMake target (either an executable or library).

Advanced users may want to link to additional targets which are exported in OSPRay’s CMake config, which includes all installed modules. All targets built with OSPRay are exported in the ospray:: namespace, therefore all targets locally used in the OSPRay source tree can be accessed from an install. For example, ospray_module_ispc can be consumed directly via the ospray::ospray_module_ispc target. All targets have their libraries, includes, and definitions attached to them for public consumption (please report bugs if this is broken!).
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 and Shutdown

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 using `ospInit` or manually instantiating a device and setting parameters on it.

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 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 "cpu" device, which maps to a fast, local CPU implementation. Other devices can also be added through additional modules, such as distributed MPI device implementations.

Once a device is created, you can call

```
void ospDeviceSetParam(OSPObject, const char *id, OSPDataType type, const void *mem);
```
Table 3.1 – Command line parameters accepted by OSPRay’s ospInit.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--osp:debug</td>
<td>enables various extra checks and debug output, and disables multi-threading</td>
</tr>
<tr>
<td>--osp:num-threads=&lt;n&gt;</td>
<td>use n threads instead of per default using all detected hardware threads</td>
</tr>
<tr>
<td>--osp:log-level=&lt;str&gt;</td>
<td>set logging level; valid values (in order of severity) are none, error, warning, info, and debug</td>
</tr>
<tr>
<td>--osp:warn-as-error</td>
<td>send warning and error messages through the error callback, otherwise send warning messages through the message callback; must have sufficient logLevel to enable warnings</td>
</tr>
<tr>
<td>--osp:verbose</td>
<td>shortcut for --osp:log-level=info and enable debug output on cout, error output on cerr</td>
</tr>
<tr>
<td>--osp:vv</td>
<td>shortcut for --osp:log-level=debug and enable debug output on cout, error output on cerr</td>
</tr>
<tr>
<td>--osp:load-modules=&lt;name&gt;[,...]</td>
<td>load one or more modules during initialization; equivalent to calling ospLoadModule(name)</td>
</tr>
<tr>
<td>--osp:log-output=&lt;dst&gt;</td>
<td>convenience for setting where status messages go; valid values for dst are cerr and cout</td>
</tr>
<tr>
<td>--osp:error-output=&lt;dst&gt;</td>
<td>convenience for setting where error messages go; valid values for dst are cerr and cout</td>
</tr>
<tr>
<td>--osp:device=&lt;name&gt;</td>
<td>use name as the type of device for OSPRay to create; e.g., --osp:device=cpu gives you the default cpu device; Note if the device to be used is defined in a module, remember to pass --osp:load-modules=&lt;name&gt; first</td>
</tr>
<tr>
<td>--osp:set-affinity=&lt;n&gt;</td>
<td>if 1, bind software threads to hardware threads; 0 disables binding; default is 1 on KNL and 0 otherwise</td>
</tr>
<tr>
<td>--osp:device-params=&lt;param&gt;:&lt;value&gt;[,...]</td>
<td>set one or more other device parameters; equivalent to calling ospDeviceSet*(param, value)</td>
</tr>
</tbody>
</table>

Table 3.2 – Parameters shared by all devices.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>numThreads</td>
<td>number of threads which OSPRay should use</td>
</tr>
<tr>
<td>string</td>
<td>logLevel</td>
<td>logging level; valid values (in order of severity) are none, error, warning, info, and debug</td>
</tr>
<tr>
<td>string</td>
<td>logOutput</td>
<td>convenience for setting where status messages go; valid values are cerr and cout</td>
</tr>
<tr>
<td>string</td>
<td>errorOutput</td>
<td>convenience for setting where error messages go; valid values are cerr and cout</td>
</tr>
<tr>
<td>bool</td>
<td>debug</td>
<td>set debug mode; equivalent to logLevel=debug and numThreads=1</td>
</tr>
<tr>
<td>bool</td>
<td>warnAsError</td>
<td>send warning and error messages through the error callback, otherwise send warning messages through the message callback; must have sufficient logLevel to enable warnings</td>
</tr>
<tr>
<td>bool</td>
<td>setAffinity</td>
<td>bind software threads to hardware threads if set to 1; 0 disables binding omitting the parameter will let OSPRay choose</td>
</tr>
</tbody>
</table>

to set parameters on the device. The semantics of setting parameters is exactly the same as ospSetParam, which is documented below in the parameters section. The following parameters can be set on all devices:

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. Note this API call will increment the ref count of the returned device handle, so applications must use ospDeviceRelease when finished using the handle to avoid leaking the underlying device object.

OSPRay allows applications to query runtime properties of a device in order to do enhanced validation of what device was loaded at runtime. The following function can be used to get these device-specific properties (attributes about the device, not parameter values)

int64_t ospDeviceGetProperty(OSPDevice, OSPDeviceProperty);

It returns an integer value of the queried property and the following properties can be provided as parameter:

OSP_DEVICE_VERSION
OSP_DEVICE_VERSION_MAJOR
OSP_DEVICE_VERSION_MINOR
OSP_DEVICE_VERSION_PATCH
OSP_DEVICE_SO_VERSION

3.1.3 Environment Variables

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_"):

Note that these environment variables take precedence over values specified through ospInit or manually set device parameters.

3.1.4 Error Handling and Status Messages

The following errors are currently used by OSPRay:

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

OSPErrror 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 (*OSPErrrorFunc)(OSPErrror, const char* errorDetails);
Table 3.3 – Environment variables interpreted by OSPRay.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPRAY_NUM_THREADS</td>
<td>equivalent to --osp:num-threads</td>
</tr>
<tr>
<td>OSPRAY_LOG_LEVEL</td>
<td>equivalent to --osp:log-level</td>
</tr>
<tr>
<td>OSPRAY_LOG_OUTPUT</td>
<td>equivalent to --osp:log-output</td>
</tr>
<tr>
<td>OSPRAY_ERROR_OUTPUT</td>
<td>equivalent to --osp:error-output</td>
</tr>
<tr>
<td>OSPRAY_DEBUG</td>
<td>equivalent to --osp:debug</td>
</tr>
<tr>
<td>OSPRAY_WARN_AS_ERROR</td>
<td>equivalent to --osp:warn-as-error</td>
</tr>
<tr>
<td>OSPRAY_SET_AFFINITY</td>
<td>equivalent to --osp:set-affinity</td>
</tr>
<tr>
<td>OSPRAY_LOAD_MODULES</td>
<td>equivalent to --osp:load-modules, can be a comma separated list of modules which will be loaded in order</td>
</tr>
<tr>
<td>OSPRAY_DEVICE</td>
<td>equivalent to --osp:device:</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSP_NO_ERROR</td>
<td>no error occurred</td>
</tr>
<tr>
<td>OSP_UNKNOWN_ERROR</td>
<td>an unknown error occurred</td>
</tr>
<tr>
<td>OSP_INVALID_ARGUMENT</td>
<td>an invalid argument was specified</td>
</tr>
<tr>
<td>OSP_INVALID_OPERATION</td>
<td>the operation is not allowed for the specified object</td>
</tr>
<tr>
<td>OSP_OUT_OF_MEMORY</td>
<td>there is not enough memory to execute the command</td>
</tr>
<tr>
<td>OSP_UNSUPPORTED_CPU</td>
<td>the CPU is not supported (minimum ISA is SSE4.1)</td>
</tr>
<tr>
<td>OSP_VERSION_MISMATCH</td>
<td>a module could not be loaded due to mismatching version</td>
</tr>
</tbody>
</table>

via

```c
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

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

in order to register a callback function of type

```c
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.

Applications can clear either callback by passing `nullptr` instead of an actual function pointer.
3.1.5 Loading OSPRay Extensions at Runtime

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

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

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

3.1.6 Shutting Down OSPRay

When the application is finished using OSPRay (typically on application exit), the OSPRay API should be finalized with

```c
void ospShutdown();
```

This API call ensures that the current device is cleaned up appropriately. Due to static object allocation having non-deterministic ordering, it is recommended that applications call `ospShutdown()` before the calling application process terminates.

3.2 Objects

All entities of OSPRay (the renderer, volumes, geometries, lights, cameras, ...) are a logical 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

```c
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 can impact performance and consistency for devices crossing a PCI bus or across a network.

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

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

This decreases its reference count and if the count reaches 0 the object will automatically get deleted. Passing `NULL` is not an error.

Sometimes applications may want to have more than one reference to an object, where it is desirable for the application to increment the reference count of an object. This is done with

```c
void ospRetain(OSPObject);
```

It is important to note that this is only necessary if the application wants to call `ospRelease` on an object more than once: objects which contain other objects as parameters internally increment/decrement ref counts and should not be explicitly done by the application.
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 (though a warning message will be posted). The following function allows adding various types of parameters with name id to a given object:

```c
void ospSetParam(OSPObject, const char *id, OSPDataType type, const void *mem);
```

The valid parameter names for all OSPObjects and what types are valid are discussed in future sections.

Note that mem must always be a pointer to the object, otherwise accidental type casting can occur. This is especially true for pointer types (OSP_VOID_PTR and OSPObject handles), as they will implicitly cast to void *, but be incorrectly interpreted. To help with some of these issues, there also exist variants of ospSetParam for specific types, such as ospSetInt and ospSetVec3f in the OSPRay utility library (found in ospray_util.h).

Users can also remove parameters that have been explicitly set from ospSetParam. 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. To remove a parameter, use

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

3.2.2 Data

OSPRay consumes data arrays from the application using a specific object type, OSPData. There are several components to describing a data array: element type, 1/2/3 dimensional striding, and whether the array is shared with the application or copied into opaque, OSPRay-owned memory.

Shared data arrays require that the application's array memory outlives the lifetime of the created OSPData, as OSPRay is referring to application memory. Where this is not preferable, applications use opaque arrays to allow the OSPData to own the lifetime of the array memory. However, opaque arrays dictate the cost of copying data into it, which should be kept in mind.

Thus the most efficient way to specify a data array from the application is to created a shared data array, which is done with

```c
OSPData ospNewSharedData(const void *sharedData,
OSPDataType,
    uint64_t numItems1,
    int64_t byteStride1 = 0,
    uint64_t numItems2 = 1,
    int64_t byteStride2 = 0,
    uint64_t numItems3 = 1,
    int64_t byteStride3 = 0);
```

The call returns an OSPData handle to the created array. The calling program guarantees that the sharedData pointer will remain valid for the duration that this data array is being used. The number of elements numItems must be positive (there cannot be an empty data object). The data is arranged in three dimensions, with specializations to two or one dimension (if some numItems are 1). The distance between consecutive elements (per dimension) is given in bytes with byteStride and can also be negative. If byteStride is zero it will be determined automatically (e.g., as sizeof(type)). Strides do not need to be
ordered, i.e., `byteStride2` can be smaller than `byteStride1`, which is equivalent to a transpose. However, if the stride should be calculated, then an ordering like `byteStride1 < byteStride2` is assumed to disambiguate.

The enum type `OSPDataType` describes the different element types that can be represented in OSPRay; valid constants are listed in the table below.

<table>
<thead>
<tr>
<th>Type/Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSP_DEVICE</td>
<td>API device object reference</td>
</tr>
<tr>
<td>OSP_DATA</td>
<td>data reference</td>
</tr>
<tr>
<td>OSP_OBJECT</td>
<td>generic object reference</td>
</tr>
<tr>
<td>OSP_CAMERA</td>
<td>camera object reference</td>
</tr>
<tr>
<td>OSP_FRAMEBUFFER</td>
<td>framebuffer object reference</td>
</tr>
<tr>
<td>OSP_LIGHT</td>
<td>light object reference</td>
</tr>
<tr>
<td>OSP_MATERIAL</td>
<td>material object reference</td>
</tr>
<tr>
<td>OSP_TEXTURE</td>
<td>texture object reference</td>
</tr>
<tr>
<td>OSP_RENDERER</td>
<td>renderer object reference</td>
</tr>
<tr>
<td>OSP_WORLD</td>
<td>world object reference</td>
</tr>
<tr>
<td>OSP_GEOMETRY</td>
<td>geometry object reference</td>
</tr>
<tr>
<td>OSP_VOLUME</td>
<td>volume object reference</td>
</tr>
<tr>
<td>OSP_TRANSFER_FUNCTION</td>
<td>transfer function object reference</td>
</tr>
<tr>
<td>OSP_IMAGE_OPERATION</td>
<td>image operation object reference</td>
</tr>
<tr>
<td>OSP_STRING</td>
<td>C-style zero-terminated character string</td>
</tr>
<tr>
<td>OSP_CHAR</td>
<td>8 bit signed character scalar</td>
</tr>
<tr>
<td>OSP_UCHAR</td>
<td>8 bit unsigned character scalar</td>
</tr>
<tr>
<td>OSP_VEC[234]UC</td>
<td>... and [234]-element vector</td>
</tr>
<tr>
<td>OSP_USHORT</td>
<td>16 bit unsigned integer scalar</td>
</tr>
<tr>
<td>OSP_INT</td>
<td>32 bit signed integer scalar</td>
</tr>
<tr>
<td>OSP_VEC[234]I</td>
<td>... and [234]-element vector</td>
</tr>
<tr>
<td>OSP_UINT</td>
<td>32 bit unsigned integer scalar</td>
</tr>
<tr>
<td>OSP_VEC[234]UI</td>
<td>... and [234]-element vector</td>
</tr>
<tr>
<td>OSP_LONG</td>
<td>64 bit signed integer scalar</td>
</tr>
<tr>
<td>OSP_VEC[234]L</td>
<td>... and [234]-element vector</td>
</tr>
<tr>
<td>OSP_UULONG</td>
<td>64 bit unsigned integer scalar</td>
</tr>
<tr>
<td>OSP_VEC[234]UL</td>
<td>... and [234]-element vector</td>
</tr>
<tr>
<td>OSP_FLOAT</td>
<td>32 bit single precision floating-point scalar</td>
</tr>
<tr>
<td>OSP_VEC[234]F</td>
<td>... and [234]-element vector</td>
</tr>
<tr>
<td>OSP_DOUBLE</td>
<td>64 bit double precision floating-point scalar</td>
</tr>
<tr>
<td>OSP_BOX[1234]I</td>
<td>32 bit integer box (lower + upper bounds)</td>
</tr>
<tr>
<td>OSP_BOX[1234]F</td>
<td>32 bit single precision floating-point box (lower + upper bounds)</td>
</tr>
<tr>
<td>OSP_LINEAR[234]F</td>
<td>32 bit single precision floating-point linear transform</td>
</tr>
<tr>
<td>OSP_AFFINE[234]F</td>
<td>32 bit single precision floating-point affine transform</td>
</tr>
<tr>
<td>OSP_VOID_PTR</td>
<td>raw memory address (only found in module extensions)</td>
</tr>
</tbody>
</table>

*Table 3.5 – Valid named constants for `OSPDataType`. 
An opaque OSPData with memory allocated by OSPRay is created with

```c
OSPData ospNewData(OSPDataType,
  uint32_t numItems1,
  uint32_t numItems2 = 1,
  uint32_t numItems3 = 1);
```

To allow for (partial) copies or updates of data arrays use

```c
void ospCopyData(const OSPData source,
  OSPData destination,
  uint32_t destinationIndex1 = 0,
  uint32_t destinationIndex2 = 0,
  uint32_t destinationIndex3 = 0);
```

which will copy the whole\(^1\) content of the source array into destination
at the given location destinationIndex. The OSPDataTypes of the data objects
must match. The region to be copied must be valid inside the destination, i.e., in
all dimensions, destinationIndex + sourceSize <= destinationSize. The
affected region \([destinationIndex, destinationIndex + sourceSize)\) is
marked as dirty, which may be used by OSPRay to only process or update that
sub-region (e.g., updating an acceleration structure). If the destination array is
shared with OSPData by the application (created with `ospNewSharedData`), then

- the source array must be shared as well (thus `ospCopyData` cannot be used
to read opaque data)
- if source and destination memory overlaps (aliasing), then behaviour is
  undefined
- except if source and destination regions are identical (including matching
  strides), which can be used by application to mark that region as dirty
  (instead of the whole OSPData)

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

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

### 3.3 Volumes

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

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

Note that OSPRay’s implementation forwards type directly to Open VKL,
allowing new Open VKL volume types to be usable within OSPRay without the
need to change (or even recompile) OSPRay.

#### 3.3.1 Structured Regular 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.

Structured regular volumes are created by passing the `structuredRegular`
type string to `ospNewVolume`. Structured volumes are represented through an
OSPData 3D array data (which may or may not be shared with the application),
where currently the voxel data needs to be laid out compact in memory in xyz-
order\(^2\)

\(^{1}\) The number of items to be copied is defined by the size of the source array

\(^{2}\) For consecutive memory addresses the x-index of the corresponding voxel changes the
quickest.
OSPRay API

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>gridOrigin</td>
<td>(0, 0, 0)</td>
<td>origin of the grid in object-space</td>
</tr>
<tr>
<td>vec3f</td>
<td>gridSpacing</td>
<td>(1, 1, 1)</td>
<td>size of the grid cells in object-space</td>
</tr>
<tr>
<td>OSPData</td>
<td>data</td>
<td></td>
<td>the actual voxel 3D data</td>
</tr>
</tbody>
</table>

Table 3.6 – Additional configuration parameters for structured regular volumes.

The parameters understood by structured volumes are summarized in the table below.

The size of the volume is inferred from the size of the 3D array data, as is the type of the voxel values (currently supported are: OSP_UCHAR, OSP_SHORT, OSP_USHORT, OSP_FLOAT, and OSP_DOUBLE).

### 3.3.2 Structured Spherical Volume

Structured spherical volumes are also supported, which are created by passing a type string of "structuredSpherical" to ospNewVolume. The grid dimensions and parameters are defined in terms of radial distance \( r \), inclination angle \( \theta \), and azimuthal angle \( \phi \), conforming with the ISO convention for spherical coordinate systems. The coordinate system and parameters understood by structured spherical volumes are summarized below.

![Coordinate system of structured spherical volumes](image)

Table 3.7 – Additional configuration parameters for structured spherical volumes.

The dimensions \( (r, \theta, \phi) \) of the volume are inferred from the size of the 3D array data, as is the type of the voxel values (currently supported are: OSP_UCHAR, OSP_SHORT, OSP_USHORT, OSP_FLOAT, and OSP_DOUBLE).

These grid parameters support flexible specification of spheres, hemispheres, spherical shells, spherical wedges, and so forth. The grid extents (computed as \([gridOrigin, gridOrigin + (dimensions - 1) * gridSpacing]) however must be constrained such that:
3.3.3 Adaptive Mesh Refinement (AMR) Volume

OSPRay currently supports block-structured (Berger-Colella) AMR volumes. Volumes are specified as a list of blocks, which exist at levels of refinement in potentially overlapping regions. Blocks exist in a tree structure, with coarser refinement level blocks containing finer blocks. The cell width is equal for all blocks at the same refinement level, though blocks at a coarser level have a larger cell width than finer levels.

There can be any number of refinement levels and any number of blocks at any level of refinement. An AMR volume type is created by passing the type string "amr" to ospNewVolume.

Blocks are defined by three parameters: their bounds, the refinement level in which they reside, and the scalar data contained within each block.

Note that cell widths are defined per refinement level, not per block.

### Table 3.8 – Additional configuration parameters for AMR volumes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPAMRMethod</td>
<td>method</td>
<td>OSP_AMR_CURRENT</td>
<td>OSPAMRMethod sampling method. Supported methods are: OSP_AMR_CURRENT, OSP_AMR_FINEST, OSP_AMR_OCTANT</td>
</tr>
<tr>
<td>float[]</td>
<td>cellWidth</td>
<td>NULL</td>
<td>array of each level’s cell width</td>
</tr>
<tr>
<td>box3f[]</td>
<td>block.bounds</td>
<td>NULL</td>
<td>data array of bounds for each AMR block</td>
</tr>
<tr>
<td>int[]</td>
<td>block.level</td>
<td>NULL</td>
<td>array of each block’s refinement level</td>
</tr>
<tr>
<td>OSPData[]</td>
<td>block.data</td>
<td>NULL</td>
<td>data array of OSPData containing the actual scalar voxel data</td>
</tr>
<tr>
<td>vec3f</td>
<td>gridOrigin</td>
<td>(0, 0, 0)</td>
<td>origin of the grid in world-space</td>
</tr>
<tr>
<td>vec3f</td>
<td>gridSpacing</td>
<td>(1, 1, 1)</td>
<td>size of the grid cells in world-space</td>
</tr>
</tbody>
</table>

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.

In particular, OSPRay’s AMR implementation was designed to cover Berger-Colella [1] and Chombo [2] AMR data. The method parameter above determines the interpolation method used when sampling the volume.

- OSP_AMR_CURRENT finds the finest refinement level at that cell and interpolates through this "current" level
- OSP_AMR_FINEST will interpolate at the closest existing cell in the volume-wide finest refinement level regardless of the sample cell’s level
- OSP_AMR_OCTANT interpolates through all available refinement levels at that cell. This method avoids discontinuities at refinement level boundaries at the cost of performance

Details and more information can be found in the publication for the implementation [3].


3. I. Wald, C. Brownlee, W. Usher, and A. Knoll. CPU volume rendering of adaptive mesh refinement data. SIGGRAPH Asia 2017 Symposium on Visualization on - SA '17, 18(8), 1–8. DOI: 10.1145/3139295.3139305

3.3.4 Unstructured Volume

Unstructured volumes can have their topology and geometry freely defined. Geometry can be composed of tetrahedral, hexahedral, wedge or pyramid cell types. The data format used is compatible with VTK and consists of multiple arrays: vertex positions and values, vertex indices, cell start indices, cell types, and cell values. An unstructured volume type is created by passing the type string "unstructured" to `ospNewVolume`.

Sampled cell values can be specified either per-vertex (`vertex.data`) or per-cell (`cell.data`). If both arrays are set, `cell.data` takes precedence.

Similar to a mesh, each cell 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, if specified. The index order for a tetrahedron is the same as VTK_TETRA: bottom triangle counterclockwise, then the top vertex.

For hexahedral cells, each hexahedron is formed by a group of eight indices into the vertices and data values. Vertex ordering is the same as VTK_HEXAHEDRON: four bottom vertices counterclockwise, then top four counterclockwise.

For wedge cells, each wedge is formed by a group of six indices into the vertices and data values. Vertex ordering is the same as VTK_WEDGE: three bottom vertices counterclockwise, then top three counterclockwise.

For pyramid cells, each cell is formed by a group of five indices into the vertices and data values. Vertex ordering is the same as VTK_PYRAMID: four bottom vertices counterclockwise, then the top vertex.

To maintain VTK data compatibility an index array may be specified via the `indexPrefixed` array that allow vertex indices to be interleaved with cell sizes in the following format: $n, id_1, ..., id_n, m, id_1, ..., id_m$.

3.3.5 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

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

The returned handle can be assigned to a volumetric model (described below) as parameter "transferFunction" using `ospSetObject`.

One type of transfer function that is supported by OSPRay is the linear transfer function, which interpolates between given equidistant colors and opacities. It is create by passing the string "piecewiseLinear" to `ospNewTransferFunction` and it is controlled by these parameters:

3.3.6 VolumetricModels

Volumes in OSPRay are given volume rendering appearance information through VolumetricModels. This decouples the physical representation of the volume (and possible acceleration structures it contains) to rendering-specific parameters (where more than one set may exist concurrently). To create a volume instance, call
### Table 3.9 – Additional configuration parameters for unstructured volumes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec3f[]</code></td>
<td>vertex.position</td>
<td>data</td>
<td>array of vertex positions</td>
</tr>
<tr>
<td><code>float[]</code></td>
<td>vertex.data</td>
<td>data</td>
<td>array of vertex data values to be sampled</td>
</tr>
<tr>
<td><code>uint32[]</code> / <code>uint64[]</code></td>
<td>index</td>
<td>data</td>
<td>array of indices (into the vertex array(s)) that form cells</td>
</tr>
<tr>
<td><code>uint32[]</code> / <code>uint64[]</code></td>
<td>indexPrefixed</td>
<td></td>
<td>alternative data array of indices compatible to VTK, where</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the indices of each cell are prefixed with the number of vertices</td>
</tr>
<tr>
<td><code>uint32[]</code> / <code>uint64[]</code></td>
<td>cell.index</td>
<td>data</td>
<td>array of locations (into the index array), specifying the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>first index of each cell</td>
</tr>
<tr>
<td><code>float[]</code></td>
<td>cell.data</td>
<td>data</td>
<td>array of cell data values to be sampled</td>
</tr>
<tr>
<td><code>uint8[]</code></td>
<td>cell.type</td>
<td>data</td>
<td>array of cell types (VTK compatible). Supported types</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>OSP_TETRAHEDRON</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>OSP_HEXAHEDRON</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>OSP_WEDGE</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>OSP_PYRAMID</code></td>
</tr>
<tr>
<td><code>bool</code></td>
<td>hexIterative</td>
<td>false</td>
<td>hexahedron interpolation method, defaults to fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>non-iterative version which could have rendering</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>inaccuracies may appear if hex is not parallelepiped</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>precomputedNormals</td>
<td>true</td>
<td>whether to accelerate by precomputing, at a cost of 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bytes/face</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec3f[]</code></td>
<td>color</td>
<td>data array of RGB colors</td>
</tr>
<tr>
<td><code>float[]</code></td>
<td>opacity</td>
<td>data array of opacities</td>
</tr>
<tr>
<td><code>vec2f</code></td>
<td>valueRange</td>
<td>domain (scalar range) this function maps from</td>
</tr>
</tbody>
</table>

### Table 3.10 – Parameters accepted by the linear transfer function.

### Table 3.11 – Parameters understood by VolumetricModel.

### 3.4 Geometries

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

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

Note that in the current implementation geometries are limited to a maximum of $2^{23}$ primitives.
### 3.4.1 Mesh

A mesh consisting of either triangles or quads is created by calling `ospNewGeometry` with type string "mesh". Once created, a mesh recognizes the following parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec3f[]</code></td>
<td>vertex.position</td>
<td>data array of vertex positions</td>
</tr>
<tr>
<td><code>vec3f[]</code></td>
<td>vertex.normal</td>
<td>data array of vertex normals</td>
</tr>
<tr>
<td><code>vec4f[] / vec3f[]</code></td>
<td>vertex.color</td>
<td>data array of vertex colors (RGBA/RGB)</td>
</tr>
<tr>
<td><code>vec2f[]</code></td>
<td>vertex.texcoord</td>
<td>data array of vertex texture coordinates</td>
</tr>
<tr>
<td><code>vec3ui[] / vec4ui[]</code></td>
<td>index</td>
<td>data array of (either triangle or quad) indices (into the vertex array(s))</td>
</tr>
</tbody>
</table>

The data type of index arrays differentiates between the underlying geometry, triangles are used for a index with `vec3ui` type and quads for `vec4ui` type. Quads are internally handled as a pair of two triangles, thus mixing triangles and quads is supported by encoding some triangle as a quad with the last two vertex indices being identical (w=z).

The `vertex.position` and `index` arrays are mandatory to create a valid mesh.

### 3.4.2 Subdivision

A mesh consisting of subdivision surfaces, created by specifying a geometry of type "subdivision". Once created, a subdivision recognizes the following parameters:

The `vertex` and `index` arrays are mandatory to create a valid subdivision surface. If no `face` array is present then a pure quad mesh is assumed (the number of indices must be a multiple of 4). Optionally supported are edge and vertex creases.

### 3.4.3 Spheres

A geometry consisting of individual spheres, each of which can have an own radius, is created by calling `ospNewGeometry` with type string "sphere". 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:

### 3.4.4 Curves

A geometry consisting of multiple curves is created by calling `ospNewGeometry` with type string "curve". The parameters defining this geometry are listed in the table below.

Depending upon the specified data type of vertex positions, the curves will be implemented Embree curves or assembled from rounded and linearly-connected segments.

Positions in `vertex.position_radius` format supports per-vertex varying radii with data type `vec4f[]` and instantiate Embree curves internally for the relevant type/basis mapping (See Embree documentation for discussion of curve types and data formatting).
### Table 3.13 – Parameters defining a Subdivision geometry.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f[]</td>
<td>vertex.position</td>
<td>NULL</td>
<td>data array of vertex positions</td>
</tr>
<tr>
<td>vec4f[]</td>
<td>vertex.color</td>
<td>NULL</td>
<td>data array of vertex colors (RGBA)</td>
</tr>
<tr>
<td>vec2f[]</td>
<td>vertex.texcoord</td>
<td>NULL</td>
<td>data array of vertex texture coordinates</td>
</tr>
<tr>
<td>float[]</td>
<td>level</td>
<td>NULL</td>
<td>5 global level of tessellation, default is 5</td>
</tr>
<tr>
<td>uint[]</td>
<td>index</td>
<td>NULL</td>
<td>data array of indices (into the vertex array(s))</td>
</tr>
<tr>
<td>float[]</td>
<td>index.level</td>
<td>NULL</td>
<td>data array of per-edge levels of tessellation, overrides global level</td>
</tr>
<tr>
<td>uint[]</td>
<td>face</td>
<td>NULL</td>
<td>data array holding the number of indices/edges (3 to 15) per face</td>
</tr>
<tr>
<td>vec2i[]</td>
<td>edgeCrease.index</td>
<td>NULL</td>
<td>data array of edge crease indices</td>
</tr>
<tr>
<td>float[]</td>
<td>edgeCrease.weight</td>
<td>NULL</td>
<td>data array of edge crease weights</td>
</tr>
<tr>
<td>uint[]</td>
<td>vertexCrease.index</td>
<td>NULL</td>
<td>data array of vertex crease indices</td>
</tr>
<tr>
<td>float[]</td>
<td>vertexCrease.weight</td>
<td>NULL</td>
<td>data array of vertex crease weights</td>
</tr>
<tr>
<td>int</td>
<td>mode</td>
<td>OSP_SUBDIVISION_SMOOTH_BOUNDARY</td>
<td>subdivision edge boundary mode. Supported modes are: OSP_SUBDIVISION_NO_BOUNDARY, OSP_SUBDIVISION_SMOOTH_BOUNDARY, OSP_SUBDIVISION_PIN_CORNERS, OSP_SUBDIVISION_PIN_BOUNDARY, OSP_SUBDIVISION_PIN_ALL</td>
</tr>
</tbody>
</table>

### Table 3.14 – Parameters defining a spheres geometry.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f[]</td>
<td>sphere.position</td>
<td></td>
<td>data array of center positions</td>
</tr>
<tr>
<td>float[]</td>
<td>sphere.radius</td>
<td>NULL</td>
<td>optional data array of the per-sphere radius</td>
</tr>
<tr>
<td>vec2f[]</td>
<td>sphere.texcoord</td>
<td>NULL</td>
<td>optional data array of texture coordinates (constant per sphere)</td>
</tr>
<tr>
<td>float</td>
<td>radius</td>
<td>0.01</td>
<td>default radius for all spheres (if sphere.radius is not set)</td>
</tr>
</tbody>
</table>

If a constant radius is used and positions are specified in a vec3f[] type of vertex.position format, then type/basis defaults to OSP_ROUND and OSP_LINEAR (this is the fastest and most memory efficient mode). Implementation is with round linear segments where each segment corresponds to a link between two vertices.

The following section describes the properties of different curve basis’ and how they use the data provided in data buffers:

**OSP_LINEAR** The indices point to the first of 2 consecutive control points in the vertex buffer. The first control point is the start and the second control point the end of the line segment. The curve goes through all control points listed in the vertex buffer.

**OSP_BEZIER** The indices point to the first of 4 consecutive control points in the vertex buffer. The first control point represents the start point of the curve, and the 4th control point the end point of the curve. The Bézier basis is interpolating, thus the curve does go exactly through the first and fourth
Table 3.15 - Parameters defining a curves geometry.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4f[]</td>
<td>vertex.position_radius</td>
<td>data array of vertex position and per-vertex radius</td>
</tr>
<tr>
<td>vec3f[]</td>
<td>vertex.position</td>
<td>data array of vertex position</td>
</tr>
<tr>
<td>float</td>
<td>radius</td>
<td>global radius of all curves (if per-vertex radius is not used), default 0.01</td>
</tr>
<tr>
<td>vec2f[]</td>
<td>vertex.texcoord</td>
<td>data array of per-vertex texture coordinates</td>
</tr>
<tr>
<td>vec4f[]</td>
<td>vertex.color</td>
<td>data array of corresponding vertex colors (RGBA)</td>
</tr>
<tr>
<td>vec3f[]</td>
<td>vertex.normal</td>
<td>data array of curve normals (only for “ribbon” curves)</td>
</tr>
<tr>
<td>uint32[]</td>
<td>index</td>
<td>data array of indices to the first vertex or tangent of a curve segment</td>
</tr>
</tbody>
</table>

- **type**: OSPCurveType for rendering the curve. Supported types are:
  - OSP_FLAT
  - OSP_ROUND
  - OSP_RIBBON

- **basis**: OSPCurveBasis for defining the curve. Supported bases are:
  - OSP_LINEAR
  - OSP_BEZIER
  - OSP_BSPLINE
  - OSP_HERMITE
  - OSP_CATMULL_ROM

control vertex.

**OSP_BSPLINE** The indices point to the first of 4 consecutive control points in the vertex buffer. This basis is not interpolating, thus the curve does in general not go through any of the control points directly. Using this basis, 3 control points can be shared for two continuous neighboring curve segments, e.g. the curves \((p_0, p_1, p_2, p_3)\) and \((p_1, p_2, p_3, p_4)\) are C1 continuous. This feature makes this basis a good choice to construct continuous multi-segment curves, as memory consumption can be kept minimal.

**OSP_HERMITE** It is necessary to have both vertex buffer and tangent buffer for using this basis. The indices point to the first of 2 consecutive points in the vertex buffer, and the first of 2 consecutive tangents in the tangent buffer. This basis is interpolating, thus does exactly go through the first and second control point, and the first order derivative at the begin and end matches exactly the value specified in the tangent buffer. When connecting two segments continuously, the end point and tangent of the previous segment can be shared.

**OSP_CATMULL_ROM** The indices point to the first of 4 consecutive control points in the vertex buffer. If \((p_0, p_1, p_2, p_3)\) represent the points then this basis goes through \(p_1\) and \(p_2\), with tangents as \((p_2 - p_0)/2\) and \((p_3 - p_1)/2\).

The following section describes the properties of different curve types and how they define the geometry of a curve:

**OSP_FLAT** This type enables faster rendering as the curve is rendered as a connected sequence of ray facing quads.

**OSP_ROUND** This type enables rendering a real geometric surface for the curve which allows closeup views. This mode renders a sweep surface by sweeping a varying radius circle tangential along the curve.

**OSP_RIBBON** The type enables normal orientation of the curve and requires a normal buffer be specified along with vertex buffer. The curve is rendered
as a flat band whose center approximately follows the provided vertex buffer and whose normal orientation approximately follows the provided normal buffer.

3.4.5 Boxes

OSPRay can directly render axis-aligned bounding boxes without the need to convert them to quads or triangles. To do so create a boxes geometry by calling `ospNewGeometry` with type string "box".

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>box3f[]</td>
<td>box</td>
<td>data array of boxes</td>
</tr>
</tbody>
</table>

3.4.6 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 “isosurface”. Each isosurface will be colored according to the transfer function assigned to the volume.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>isovalue</td>
<td>single isovalue</td>
</tr>
<tr>
<td>float[]</td>
<td>isovalue</td>
<td>data array of isovalue</td>
</tr>
<tr>
<td>OSPVolumetricModel</td>
<td>volume</td>
<td>handle of the VolumetricModels to be isosurfaced</td>
</tr>
</tbody>
</table>

3.4.7 GeometricModels

Geometries are matched with surface appearance information through GeometricModels. These take a geometry, which defines the surface representation, and applies either full-object or per-primitive color and material information. To create a geometric model, call

```c
OSPGeometricModel ospNewGeometricModel(OSPGeometry geometry);
```

Color and material are fetched with the primitive ID of the hit (clamped to the valid range, thus a single color or material is fine), or mapped first via the index array (if present). All parameters are optional, however, some renderers (notably the path tracer) require a material to be set. Materials are either handles of `OSPMaterial`, or indices into the material array on the renderer, which allows to build a world which can be used by different types of renderers.

3.5 Lights

To create a new light source of given type use

```c
OSPLight ospNewLight(const char *type);
```

All light sources accept the following parameters:

- The following light types are supported by most OSPRay renderers:
### Table 3.18 – Parameters understood by GeometricModel.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPMaterial / uint32</td>
<td>material</td>
<td>optional material applied to the geometry, may be an index into the material parameter on the renderer (if it exists)</td>
</tr>
<tr>
<td>vec4f</td>
<td>color</td>
<td>optional color assigned to the geometry</td>
</tr>
<tr>
<td>OSPMaterial[] / uint32[]</td>
<td>material data</td>
<td>optional data array of (per-primitive) materials, may be an index into the material parameter on the renderer (if it exists)</td>
</tr>
<tr>
<td>vec4f[]</td>
<td>color data</td>
<td>optional data array of (per-primitive) colors</td>
</tr>
<tr>
<td>uint8[]</td>
<td>index data</td>
<td>optional data array of per-primitive indices into color and material</td>
</tr>
</tbody>
</table>

### Table 3.19 – Parameters accepted by all lights.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>color</td>
<td>white</td>
<td>color of the light</td>
</tr>
<tr>
<td>float</td>
<td>intensity</td>
<td>1</td>
<td>intensity of the light (a factor)</td>
</tr>
<tr>
<td>bool</td>
<td>visible</td>
<td>true</td>
<td>whether the light can be directly seen</td>
</tr>
</tbody>
</table>

### 3.5.1 Directional Light / Distant Light

The distant light (or traditionally the directional light) is thought to be 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>direction</td>
<td>main emission direction of the distant light</td>
</tr>
<tr>
<td>float</td>
<td>angularDiameter</td>
<td>apparent size (angle in degree) of the light</td>
</tr>
</tbody>
</table>

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°.

### 3.5.2 Point Light / Sphere Light

The sphere light (or the special case point light) is a light emitting uniformly in all directions from the surface towards the outside. It does not emit any light towards the inside of the sphere. 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>position</td>
<td>the center of the sphere light, in world-space</td>
</tr>
<tr>
<td>float</td>
<td>radius</td>
<td>the size of the sphere light</td>
</tr>
</tbody>
</table>

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

The spotlight 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 spotlight supports the special parameters listed in the table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>position</td>
<td>(0, 0, 0)</td>
<td>the center of the spotlight, in world-space</td>
</tr>
<tr>
<td>vec3f</td>
<td>direction</td>
<td>(0, 0, 1)</td>
<td>main emission direction of the spot</td>
</tr>
<tr>
<td>float</td>
<td>openingAngle</td>
<td>180</td>
<td>full opening angle (in degree) of the spot; outside of this cone is no illumination</td>
</tr>
<tr>
<td>float</td>
<td>penumbraAngle</td>
<td>5</td>
<td>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</td>
</tr>
<tr>
<td>float</td>
<td>radius</td>
<td>0</td>
<td>the size of the spotlight, the radius of a disk with normal direction</td>
</tr>
</tbody>
</table>

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 Quad Light

The quad\(^3\) 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 quad light supports the following special parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>position</td>
<td>world-space position of one vertex of the quad light</td>
</tr>
<tr>
<td>vec3f</td>
<td>edge1</td>
<td>vector to one adjacent vertex</td>
</tr>
<tr>
<td>vec3f</td>
<td>edge2</td>
<td>vector to the other adjacent vertex</td>
</tr>
</tbody>
</table>

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.

\(^3\) actually a parallelogram
3.5.5 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 general parameters the HDRI light supports the following special parameters:

Table 3.24 – Special parameters accepted by the HDRI light.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>up</td>
<td>up direction of the light in world-space</td>
</tr>
<tr>
<td>vec3f</td>
<td>direction</td>
<td>direction to which the center of the texture will be mapped to (analog to panoramic camera)</td>
</tr>
<tr>
<td>OSPTexture</td>
<td>map</td>
<td>environment map in latitude / longitude format</td>
</tr>
</tbody>
</table>

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

3.5.6 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).

3.5.7 Emissive Objects

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

3.6 Scene Hierarchy

3.6.1 Groups

Groups in OSPRay represent collections of GeometricModels and VolumetricModels which share a common local-space coordinate system. To create a group call...
OSPGroup ospNewGroup();

Groups take arrays of geometric models and volumetric models, but they are optional. In other words, there is no need to create empty arrays if there are no geometries or volumes in the group.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPGeometricModel[]</td>
<td>geometry</td>
<td>NULL</td>
<td>data array of GeometricModels</td>
</tr>
<tr>
<td>OSPVolumetricModel[]</td>
<td>volume</td>
<td>NULL</td>
<td>data array of VolumetricModels</td>
</tr>
<tr>
<td>bool</td>
<td>dynamicScene</td>
<td>false</td>
<td>use RTC_SCENE_DYNAMIC flag (faster BVH build, slower ray traversal), otherwise uses RTC_SCENE_STATIC flag (faster ray traversal, slightly slower BVH build)</td>
</tr>
<tr>
<td>bool</td>
<td>compactMode</td>
<td>false</td>
<td>tell Embree to use a more compact BVH in memory by trading ray traversal performance</td>
</tr>
<tr>
<td>bool</td>
<td>robustMode</td>
<td>false</td>
<td>tell Embree to enable more robust ray intersection code paths (slightly slower)</td>
</tr>
</tbody>
</table>

Note that groups only need to re-committed if a geometry or volume changes (surface/scalar field representation). Appearance information on OSPGeometricModel and OSPVolumetricModel can be changed freely, as internal acceleration structures do not need to be reconstructed.

### 3.6.2 Instances

Instances in OSPRay represent a single group’s placement into the world via a transform. To create and instance call

OSPInstance ospNewInstance(OSPGroup);

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>affine3f</td>
<td>xfm</td>
<td>(identity)</td>
<td>world-space transform for all attached geometries and volumes</td>
</tr>
</tbody>
</table>

### 3.6.3 World

Worlds are a container of scene data represented by instances. To create an (empty) world call

OSPWorld ospNewWorld();

Objects are placed in the world through an array of instances. Similar to [group], the array of instances is optional: there is no need to create empty arrays if there are no instances (though there will be nothing to render).

Applications can query the world (axis-aligned) bounding box after the world has been committed. To get this information, call

OSPBounds ospGetBounds(OSPObject);

This call can also take OSPGroup and OSPInstance as well: all other object types will return an empty bounding box.

Finally, Worlds can be configured with parameters for making various feature/performance trade-offs (similar to groups).
Table 3.27 – Parameters understood by worlds.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPInstance[]</td>
<td>instance</td>
<td>NULL</td>
<td>data array with handles of the instances</td>
</tr>
<tr>
<td>OSPLight[]</td>
<td>light</td>
<td>NULL</td>
<td>data array with handles of the lights</td>
</tr>
<tr>
<td>bool</td>
<td>dynamicScene</td>
<td>false</td>
<td>use RTC_SCENE_DYNAMIC flag (faster BVH build, slower ray traversal), otherwise uses RTC_SCENE_STATIC flag (faster ray traversal, slightly slower BVH build)</td>
</tr>
<tr>
<td>bool</td>
<td>compactMode</td>
<td>false</td>
<td>tell Embree to use a more compact BVH in memory by trading ray traversal performance</td>
</tr>
<tr>
<td>bool</td>
<td>robustMode</td>
<td>false</td>
<td>tell Embree to enable more robust ray intersection code paths (slightly slower)</td>
</tr>
</tbody>
</table>

3.7 Renderers

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`

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

General parameters of all renderers are

Table 3.28 – Parameters understood by all renderers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>pixelSamples</td>
<td>1</td>
<td>samples per pixel</td>
</tr>
<tr>
<td>int</td>
<td>maxPathLength</td>
<td>20</td>
<td>maximum ray recursion depth</td>
</tr>
<tr>
<td>float</td>
<td>minContribution</td>
<td>0.001</td>
<td>sample contributions below this value will be neglected to speedup rendering</td>
</tr>
<tr>
<td>float / vec3f / vec4f</td>
<td>varianceThreshold</td>
<td>0</td>
<td>threshold for adaptive accumulation</td>
</tr>
<tr>
<td>OSPTexture</td>
<td>map_backplate</td>
<td>black, transparent</td>
<td>background color and alpha (RGBA), if no map_backplate is set</td>
</tr>
<tr>
<td>OSPTexture</td>
<td>map_maxDepth</td>
<td>optional texture image used as background</td>
<td></td>
</tr>
<tr>
<td>OSPMaterial[]</td>
<td>material</td>
<td>optional data array of materials which can be indexed by a GeometricModel's material parameter</td>
<td></td>
</tr>
</tbody>
</table>

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.

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 eases transparency-aware blending of the image with an arbitrary background image by the application. The parameter backgroundColor or map_backplate can be used to already blend with a constant background color or backplate texture, respectively, (and alpha) during rendering.
OSPRay API

OSPRay renderers support 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 `map_maxDepth` 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.

### 3.7.1 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” to `ospNewRenderer`. In addition to the general parameters understood by all renderers, the SciVis renderer supports the following parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>aoSamples</td>
<td>0</td>
<td>number of rays per sample to compute ambient occlusion</td>
</tr>
<tr>
<td>float</td>
<td>aoRadius</td>
<td>10^20</td>
<td>maximum distance to consider for ambient occlusion</td>
</tr>
<tr>
<td>float</td>
<td>aoIntensity</td>
<td>1</td>
<td>ambient occlusion strength</td>
</tr>
<tr>
<td>float</td>
<td>volumeSamplingRate</td>
<td>1</td>
<td>sampling rate for volumes</td>
</tr>
</tbody>
</table>

### 3.7.2 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>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>geometryLights</td>
<td>true</td>
<td>whether to render light emitted from geometries</td>
</tr>
<tr>
<td>int</td>
<td>roulettePathLength</td>
<td>5</td>
<td>ray recursion depth at which to start Russian roulette termination</td>
</tr>
<tr>
<td>float</td>
<td>maxContribution</td>
<td>(\infty)</td>
<td>samples are clamped to this value before they are accumulated into the framebuffer</td>
</tr>
</tbody>
</table>

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

The path tracer supports volumes with multiple scattering. The scattering albedo can be specified using the transfer function. Extinction is assumed to be spectrally constant.

### 3.7.3 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

```c
OSPMaterial ospNewMaterial(const char *renderer_type, const char *material_type);
```

The returned handle can then be used to assign the material to a given geometry with
void ospSetObject(OSPGeometricModel, "material", OSPMaterial);

### 3.7.3.1 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 "obj" to ospNewMaterial. Its main parameters are

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>kd</td>
<td>white 0.8</td>
<td>diffuse color</td>
</tr>
<tr>
<td>vec3f</td>
<td>ks</td>
<td>black</td>
<td>specular color</td>
</tr>
<tr>
<td>float</td>
<td>ns</td>
<td>10</td>
<td>shininess (Phong exponent), usually in [2–10^4]</td>
</tr>
<tr>
<td>float</td>
<td>d</td>
<td>opaque</td>
<td>opacity</td>
</tr>
<tr>
<td>vec3f</td>
<td>tf</td>
<td>black</td>
<td>transparency filter color</td>
</tr>
<tr>
<td>OSPTexture</td>
<td>map_bump</td>
<td>NULL</td>
<td>normal map</td>
</tr>
</tbody>
</table>

Table 3.31 – 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 $K_d$, $K_s$, and $T_f$ 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 $K_d$ 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, as can be seen in the figure below).

![Figure 3.5](image) – Comparison of diffuse rooms with 100% reflecting white paint (left) and realistic 80% reflecting white paint (right), which leads to higher overall contrast. Note that exposure has been adjusted to achieve similar brightness levels.

If present, the color component of geometries is also used for the diffuse color $K_d$ and the alpha component is also used for the opacity $d$.

Note that currently only the path tracer implements colored transparency with $T_f$.

Normal mapping can simulate small geometric features via the texture map_bump. The normals $n$ in the normal map are with respect to the local tangential shading coordinate system and are encoded as $\frac{1}{2}(n + 1)$, thus a texel $(0.5, 0.5, 1)^4$ 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

---

4 respectively $(127, 127, 255)$ for 8 bit textures.
look green toward 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.

Figure 3.6 – Normal map representing an exalted square pyramidal frustum.

All parameters (except $T_f$) can be textured by passing a texture handle, pre-
fixed 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 3.7 – Rendering of a OBJ material with wood textures.

3.7.3.2 Principled

The Principled material is the most complex material offered by the path tracer,
which is capable of producing a wide variety of materials (e.g., plastic, metal,
wood, glass) by combining multiple different layers and lobes. It uses the GGX
microfacet distribution with approximate multiple scattering for dielectrics and
metals, uses the Oren-Nayar model for diffuse reflection, and is energy conserv-
ing. To create a Principled material, pass the type string "principled" to osp-
NewMaterial. Its parameters are listed in the table below.

All parameters can be textured by passing a texture handle, prefixed with
"map_" (e.g., "map_baseColor"). texture transformations are supported as well.

3.7.3.3 CarPaint

The CarPaint material is a specialized version of the Principled material for ren-
dering different types of car paints. To create a CarPaint material, pass the type
string "carPaint" to ospNewMaterial. Its parameters are listed in the table be-
low.
### Table 3.32 – Parameters of the Principled material.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>baseColor</td>
<td>white 0.8</td>
<td>base reflectivity (diffuse and/or metallic)</td>
</tr>
<tr>
<td>vec3f</td>
<td>edgeColor</td>
<td>white</td>
<td>edge tint (metallic only)</td>
</tr>
<tr>
<td>float</td>
<td>metallic</td>
<td>0</td>
<td>mix between dielectric (diffuse and/or specular) and metallic (specular only with complex IOR) in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>diffuse</td>
<td>1</td>
<td>diffuse reflection weight in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>specular</td>
<td>1</td>
<td>specular reflection/transmission weight in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>ior</td>
<td>1</td>
<td>dielectric index of refraction</td>
</tr>
<tr>
<td>float</td>
<td>transmission</td>
<td>0</td>
<td>specular transmission weight in [0–1]</td>
</tr>
<tr>
<td>vec3f</td>
<td>transmissionColor</td>
<td>white</td>
<td>attenuated color due to transmission (Beer’s law)</td>
</tr>
<tr>
<td>float</td>
<td>transmissionDepth</td>
<td>1</td>
<td>distance at which color attenuation is equal to transmissionColor</td>
</tr>
<tr>
<td>float</td>
<td>roughness</td>
<td>0</td>
<td>diffuse and specular roughness in [0–1], 0 is perfectly smooth</td>
</tr>
<tr>
<td>float</td>
<td>anisotropy</td>
<td>0</td>
<td>amount of specular anisotropy in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>rotation</td>
<td>0</td>
<td>rotation of the direction of anisotropy in [0–1], 1 is going full circle</td>
</tr>
<tr>
<td>float</td>
<td>normal</td>
<td>1</td>
<td>default normal map/scale for all layers</td>
</tr>
<tr>
<td>float</td>
<td>baseNormal</td>
<td>1</td>
<td>base normal map/scale (overrides default normal)</td>
</tr>
<tr>
<td>bool</td>
<td>thin</td>
<td>false</td>
<td>flag specifying whether the material is thin or solid</td>
</tr>
<tr>
<td>float</td>
<td>thickness</td>
<td>1</td>
<td>thickness of the material (thin only), affects the amount of color attenuation due to specular transmission</td>
</tr>
<tr>
<td>float</td>
<td>backlight</td>
<td>0</td>
<td>amount of diffuse transmission (thin only) in [0–2], 1 is 50% reflection and 50% transmission, 2 is transmission only</td>
</tr>
<tr>
<td>float</td>
<td>coat</td>
<td>0</td>
<td>clear coat layer weight in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>coatIor</td>
<td>1.5</td>
<td>clear coat index of refraction</td>
</tr>
<tr>
<td>vec3f</td>
<td>coatColor</td>
<td>white</td>
<td>clear coat color tint</td>
</tr>
<tr>
<td>float</td>
<td>coatThickness</td>
<td>1</td>
<td>clear coat thickness, affects the amount of color attenuation</td>
</tr>
<tr>
<td>float</td>
<td>coatRoughness</td>
<td>0</td>
<td>clear coat roughness in [0–1], 0 is perfectly smooth</td>
</tr>
<tr>
<td>float</td>
<td>coatNormal</td>
<td>1</td>
<td>clear coat normal map/scale (overrides default normal)</td>
</tr>
<tr>
<td>float</td>
<td>sheen</td>
<td>0</td>
<td>sheen layer weight in [0–1]</td>
</tr>
<tr>
<td>vec3f</td>
<td>sheenColor</td>
<td>white</td>
<td>sheen color tint</td>
</tr>
<tr>
<td>float</td>
<td>sheenTint</td>
<td>0</td>
<td>how much sheen is tinted from sheenColor toward baseColor</td>
</tr>
<tr>
<td>float</td>
<td>sheenRoughness</td>
<td>0.2</td>
<td>sheen roughness in [0–1], 0 is perfectly smooth</td>
</tr>
<tr>
<td>float</td>
<td>opacity</td>
<td>1</td>
<td>cut-out opacity/transparency, 1 is fully opaque</td>
</tr>
</tbody>
</table>

All parameters can be textured by passing a texture handle, prefixed with “map_” (e.g., “map_baseColor”). Texture transformations are supported as well.

#### 3.7.3.4 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

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 counter-intuitive but can be found in published measurements. For accuracy the index of refraction can be given as an array of spectral samples in \( \text{ior} \), each sample a triplet of wavelength (in nm),...
Figure 3.8 – Rendering of a Principled coated brushed metal material with textured anisotropic rotation and a dust layer (sheen) on top.

Table 3.33 – Parameters of the CarPaint material.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>baseColor</td>
<td>white 0.8</td>
<td>diffuse base reflectivity</td>
</tr>
<tr>
<td>float</td>
<td>roughness</td>
<td>0</td>
<td>diffuse roughness in [0–1], 0 is perfectly smooth</td>
</tr>
<tr>
<td>float</td>
<td>normal</td>
<td>1</td>
<td>normal map/scale</td>
</tr>
<tr>
<td>float</td>
<td>flakeDensity</td>
<td>0</td>
<td>density of metallic flakes in [0–1], 0 disables flakes, 1 fully covers the surface with flakes</td>
</tr>
<tr>
<td>float</td>
<td>flakeScale</td>
<td>100</td>
<td>scale of the flake structure, higher values increase the amount of flakes</td>
</tr>
<tr>
<td>float</td>
<td>flakeSpread</td>
<td>0.3</td>
<td>flake spread in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>flakeJitter</td>
<td>0.75</td>
<td>flake randomness in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>flakeRoughness</td>
<td>0.3</td>
<td>flake roughness in [0–1], 0 is perfectly smooth</td>
</tr>
<tr>
<td>float</td>
<td>coat</td>
<td>1</td>
<td>clear coat layer weight in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>coatIor</td>
<td>1.5</td>
<td>clear coat index of refraction</td>
</tr>
<tr>
<td>vec3f</td>
<td>coatColor</td>
<td>white</td>
<td>clear coat color tint</td>
</tr>
<tr>
<td>float</td>
<td>coatThickness</td>
<td>1</td>
<td>clear coat thickness, affects the amount of color attenuation</td>
</tr>
<tr>
<td>float</td>
<td>coatRoughness</td>
<td>0</td>
<td>clear coat roughness in [0–1], 0 is perfectly smooth</td>
</tr>
<tr>
<td>float</td>
<td>coatNormal</td>
<td>1</td>
<td>clear coat normal map/scale</td>
</tr>
<tr>
<td>vec3f</td>
<td>flipflopColor</td>
<td>white</td>
<td>reflectivity of coated flakes at grazing angle, used together with coatColor produces a pearslecent paint</td>
</tr>
<tr>
<td>float</td>
<td>flipflopFalloff</td>
<td>1</td>
<td>flip flop color falloff, 1 disables the flip flop effect</td>
</tr>
</tbody>
</table>

Table 3.34 – Parameters of the Metal material.

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

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.
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 notable edging effects.

3.7.3.5 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>color</td>
<td>white 0.9</td>
<td>reflectivity at normal incidence (0 degree)</td>
</tr>
<tr>
<td>vec3f</td>
<td>edgeColor</td>
<td>white</td>
<td>reflectivity at grazing angle (90 degree)</td>
</tr>
<tr>
<td>float</td>
<td>roughness</td>
<td>0.1</td>
<td>roughness, in [0–1], 0 is perfect mirror</td>
</tr>
</tbody>
</table>

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). If present, the color component of geometries is also used for reflectivity at normal incidence color. 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.

3.7.3.6 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>eta</td>
<td>1.5</td>
<td>index of refraction</td>
</tr>
<tr>
<td>vec3f</td>
<td>attenuationColor</td>
<td>white</td>
<td>resulting color due to attenuation</td>
</tr>
<tr>
<td>float</td>
<td>attenuationDistance</td>
<td>1</td>
<td>distance affecting attenuation</td>
</tr>
</tbody>
</table>

For convenience, the rather counter-intuitive 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.

Figure 3.11 – Rendering of a fictional Alloy material with textured color.

Figure 3.12 – Rendering of a Glass material with orange attenuation.
3.7.3.7 ThinGlass

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>eta</td>
<td>1.5</td>
<td>index of refraction</td>
</tr>
<tr>
<td>vec3f</td>
<td>attenuationColor</td>
<td>white</td>
<td>resulting color due to attenuation</td>
</tr>
<tr>
<td>float</td>
<td>attenuationDistance</td>
<td>1</td>
<td>distance affecting attenuation</td>
</tr>
<tr>
<td>float</td>
<td>thickness</td>
<td>1</td>
<td>virtual thickness</td>
</tr>
</tbody>
</table>

Table 3.38 – 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). If present, the color component of geometries is also used for the attenuation color. 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 3.13 – Rendering of a ThinGlass material with red attenuation.

Figure 3.14 – Example image of a colored window made with textured attenuation of the ThinGlass material.
### 3.7.3.8 MetallicPaint

The *path tracer* offers a metallic paint material, consisting of a base coat with optional flakes and a clear coat. To create a MetallicPaint material pass the type string "metallicPaint" to `ospNewMaterial`. Its parameters are listed in the table below.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>baseColor</td>
<td>white</td>
<td>color of base coat</td>
</tr>
<tr>
<td>float</td>
<td>flakeAmount</td>
<td>0.3</td>
<td>amount of flakes, in [0–1]</td>
</tr>
<tr>
<td>vec3f</td>
<td>flakeColor</td>
<td>Aluminium</td>
<td>color of metallic flakes</td>
</tr>
<tr>
<td>float</td>
<td>flakeSpread</td>
<td>0.5</td>
<td>spread of flakes, in [0–1]</td>
</tr>
<tr>
<td>float</td>
<td>eta</td>
<td>1.5</td>
<td>index of refraction of clear coat</td>
</tr>
</tbody>
</table>

The color of the base coat `baseColor` can be textured by a texture map `baseColor`, which also supports texture transformations. If present, the color component of `geometries` is also used for the color of the base coat. Parameter `flakeAmount` controls the proportion of flakes in the base coat, so when setting it to 1 the `baseColor` will not be visible. The shininess of the metallic component is governed by `flakeSpread`, which controls the variation of the orientation of the flakes, similar to the `roughness` parameter of `Metal`. Note that the effect of the metallic flakes is currently only computed on average, thus individual flakes are not visible.

![MetallicPaint material rendering](image)

Table 3.39 – Parameters of the Metallic-Paint material.

### 3.7.3.9 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`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec3f</td>
<td>color</td>
<td>white</td>
<td>color of the emitted light</td>
</tr>
<tr>
<td>float</td>
<td>intensity</td>
<td>1</td>
<td>intensity of the light (a factor)</td>
</tr>
<tr>
<td>float</td>
<td>transparency</td>
<td>1</td>
<td>material transparency</td>
</tr>
</tbody>
</table>

![Luminous material rendering](image)

Table 3.40 – Parameters accepted by the Luminous material.
3.7.4 Texture

OSPRay currently implements two texture types (texture2d and volume) and is open for extension to other types by applications. More types may be added in future releases.

To create a new texture use

\[
\text{OSPTexture } \text{ospNewTexture(const char *type);}
\]

3.7.4.1 Texture2D

The texture2d texture type implements an image-based texture, where its parameters are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>format</td>
<td>OSPTextureFormat for the texture</td>
</tr>
<tr>
<td>int</td>
<td>filter</td>
<td>default OSP_TEXTURE_FILTER_BILINEAR, alternatively OSP_TEXTURE_FILTER_NEAREST</td>
</tr>
<tr>
<td>OSPData</td>
<td>data</td>
<td>the actual texel 2D data</td>
</tr>
</tbody>
</table>

The supported texture formats for texture2d are:

The size of the texture is inferred from the size of the 2D array data, which also needs have a compatible type to format. The texel data in data starts with the texels in the lower left corner of the texture image, like in OpenGL. Per default a texture fetch is filtered by performing bi-linear interpolation of the nearest 2x2 texels; if instead fetching only the nearest texel is desired (i.e., no filtering) then pass the OSP_TEXTURE_FILTER_NEAREST flag.

3.7.4.2 TextureVolume

The volume texture type implements texture lookups based on 3D world coordinates of the surface hit point on the associated geometry. If the given hit point is within the attached volume, the volume is sampled and classified with the transfer function attached to the volume. This implements the ability to visualize volume values (as colored by its transfer function) on arbitrary surfaces inside the volume (as opposed to an isosurface showing a particular value in the volume). Its parameters are as follows:

TextureVolume can be used for implementing slicing of volumes with any geometry type. It enables coloring of the slicing geometry with a different transfer function than that of the sliced volume.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSP_TEXTURE_RGBA8</td>
<td>8 bit [0–255] linear components red, green, blue, alpha</td>
</tr>
<tr>
<td>OSP_TEXTURE_SRGBA</td>
<td>8 bit sRGB gamma encoded color components, and linear alpha</td>
</tr>
<tr>
<td>OSP_TEXTURE_RGBA32F</td>
<td>32 bit float components red, green, blue, alpha</td>
</tr>
<tr>
<td>OSP_TEXTURE_RGB8</td>
<td>8 bit [0–255] linear components red, green, blue</td>
</tr>
<tr>
<td>OSP_TEXTURE_SRGB</td>
<td>8 bit sRGB gamma encoded components red, green, blue</td>
</tr>
<tr>
<td>OSP_TEXTURE_RGB32F</td>
<td>32 bit float components red, green, blue</td>
</tr>
<tr>
<td>OSP_TEXTURE_R8</td>
<td>8 bit [0–255] linear single component</td>
</tr>
<tr>
<td>OSP_TEXTURE_RA8</td>
<td>8 bit [0–255] linear two component</td>
</tr>
<tr>
<td>OSP_TEXTURE_L8</td>
<td>8 bit [0–255] gamma encoded luminance</td>
</tr>
<tr>
<td>OSP_TEXTURE_LA8</td>
<td>8 bit [0–255] gamma encoded luminance, and linear alpha</td>
</tr>
<tr>
<td>OSP_TEXTURE_R32F</td>
<td>32 bit float single component</td>
</tr>
</tbody>
</table>

Table 3.42 – Supported texture formats by texture2d, i.e., valid constants of type OSPTextureFormat.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPVolume</td>
<td>volume</td>
<td>volume used to generate color lookups</td>
</tr>
</tbody>
</table>

3.7.5 Texture2D 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4f</td>
<td>transform</td>
<td>interpreted as 2x2 matrix (linear part), column-major</td>
</tr>
<tr>
<td>float</td>
<td>rotation</td>
<td>angle in degree, counterclockwise, around center</td>
</tr>
<tr>
<td>vec2f</td>
<td>scale</td>
<td>enlarge texture, relative to center (0.5, 0.5)</td>
</tr>
<tr>
<td>vec2f</td>
<td>translation</td>
<td>move texture in positive direction (right/up)</td>
</tr>
</tbody>
</table>

Table 3.43 – Parameters of volume texture type.

Table 3.44 – Parameters to define texture coordinate transformations.

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).

3.7.6 Cameras

To create a new camera of given type use

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

All cameras accept these parameters:

The camera is placed and oriented in the world with position, direction 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
**Type** | **Name** | **Description**
--- | --- | ---
vec3f | position | position of the camera in world-space
vec3f | direction | main viewing direction of the camera
vec3f | 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 3.45 – Parameters accepted by all cameras.

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.

### 3.7.6.1 Perspective Camera

The perspective camera implements a simple thin lens 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 3.46 – Parameters accepted by the perspective camera.

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

Note that when computing the aspect ratio a potentially set image region (using imageStart & imageEnd) needs to be regarded as well.

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.

### 3.7.6.2 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
Figure 3.17 – Example image created with the perspective camera, featuring depth of field.

Figure 3.18 – Enabling the architectural flag corrects the perspective projection distortion, resulting in parallel vertical edges.
type string “orthographic” to ospNewCamera. In addition to the general parameters understood by all cameras the orthographic camera supports the following special parameters:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>height</td>
<td>size of the camera’s image plane in y, in world coordinates</td>
</tr>
<tr>
<td>float</td>
<td>aspect</td>
<td>ratio of width by height of the frame</td>
</tr>
</tbody>
</table>

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.

### 3.7.6.3 Panoramic Camera

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

### 3.7.7 Picking

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

```c
void ospPick(OSPPickResult *,
             OSPFrameBuffer,
             OSPRenderer,
             OSPCamera,
             OSPWorld,
             osp_vec2f screenPos);
```

The result is returned in the provided OSPPickResult struct.
Figure 3.20 – Example image created with the orthographic camera.

Figure 3.21 – Latitude / longitude map created with the panoramic camera.
typedef struct {
    int hasHit;
    osp_vec3f worldPosition;
    OSPGeometricModel GeometricModel;
    uint32_t primID;
} 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.

3.8 Framebuffer

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

```c
OSPFrameBuffer ospNewFrameBuffer(osp_vec2i size,
    OSPFrameBufferFormat format = OSP_FB_SRGBA,
    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:

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

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSP_FB_COLOR</td>
<td>RGB color including alpha</td>
</tr>
<tr>
<td>OSP_FB_DEPTH</td>
<td>euclidean distance to the camera (not to the image plane), as linear 32 bit float</td>
</tr>
<tr>
<td>OSP_FB_ACCUM</td>
<td>accumulation buffer for progressive refinement</td>
</tr>
<tr>
<td>OSP_FB_VARIANCE</td>
<td>for estimation of the current noise level if OSP_FB_ACCUM is also present, see rendering</td>
</tr>
<tr>
<td>OSP_FB_NORMAL</td>
<td>accumulated world-space normal of the first hit, as vec3f</td>
</tr>
<tr>
<td>OSP_FB_ALBEDO</td>
<td>accumulated material albedo (color without illumination) at the first hit, as vec3f</td>
</tr>
</tbody>
</table>

If a certain channel value is not specified, the given buffer channel will not be present. Note that OSPRay makes a clear distinction between the external
format of the framebuffer and the internal one: The external format is the 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 `OSPImageOperation` image operation.

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

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

Note that OSP_FB_ACCUM or OSP_FB_VARIANCE cannot 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

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

The individual channels of a framebuffer can be cleared with

```c
void ospResetAccumulation(OSPFrameBuffer);
```

This function will clear all accumulating buffers (OSP_FB_VARIANCE, OSP_FB_NORMAL, and OSP_FB_ALBEDO, if present) and resets the accumulation counter `accumID`. It is unspecified if the existing color and depth buffers are physically cleared when `ospResetAccumulation` is called.

If OSP_FB_VARIANCE is specified, an estimate of the variance of the last accumulated frame can be queried with

```c
float ospGetVariance(OSPFrameBuffer);
```

Note this value is only updated after synchronizing with OSP_FRAME_FINISHED, as further described in asynchronous rendering.

The framebuffer takes a list of pixel operations to be applied to the image in sequence as an `OSPData`. The pixel operations will be run in the order they are in the array.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPImageOperation[]</td>
<td>imageOperation</td>
<td>ordered sequence of image operations</td>
</tr>
</tbody>
</table>

Table 3.50 – Parameters accepted by the framebuffer.

### 3.8.1 Image Operation

Image operations are functions that are applied to every pixel of a frame. 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

```c
OSPImageOperation ospNewImageOperation(const char *type);
```
### 3.8.1.1 Tone Mapper

The tone mapper is a pixel operation which implements a generic filmic tone mapping operator. Using the default parameters it approximates the Academy Color Encoding System (ACES). The tone mapper is created by passing the type string “tonemapper” to `ospNewImageOperation`. The tone mapping curve can be customized using the parameters listed in the table below.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>exposure</td>
<td>1.0</td>
<td>amount of light per unit area</td>
</tr>
<tr>
<td>float</td>
<td>contrast</td>
<td>1.6773</td>
<td>contrast (toe of the curve); typically is in [1–2]</td>
</tr>
<tr>
<td>float</td>
<td>shoulder</td>
<td>0.9714</td>
<td>highlight compression (shoulder of the curve); typically is in [0.9–1]</td>
</tr>
<tr>
<td>float</td>
<td>midIn</td>
<td>0.18</td>
<td>mid-level anchor input; default is 18% gray</td>
</tr>
<tr>
<td>float</td>
<td>midOut</td>
<td>0.18</td>
<td>mid-level anchor output; default is 18% gray</td>
</tr>
<tr>
<td>float</td>
<td>hdrMax</td>
<td>11.0785</td>
<td>maximum HDR input that is not clipped</td>
</tr>
<tr>
<td>bool</td>
<td>acesColor</td>
<td>true</td>
<td>apply the ACES color transforms</td>
</tr>
</tbody>
</table>

To use the popular “Uncharted 2” filmic tone mapping curve instead, set the parameters to the values listed in the table below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrast</td>
<td>1.1759</td>
</tr>
<tr>
<td>shoulder</td>
<td>0.9746</td>
</tr>
<tr>
<td>midIn</td>
<td>0.18</td>
</tr>
<tr>
<td>midOut</td>
<td>0.18</td>
</tr>
<tr>
<td>hdrMax</td>
<td>6.3704</td>
</tr>
<tr>
<td>acesColor</td>
<td>false</td>
</tr>
</tbody>
</table>

### 3.8.1.2 Denoiser

OSPRay comes with a module that adds support for Intel® Open Image Denoise. This is provided as an optional module as it creates an additional project dependency at compile time. The module implements a “denoiser” frame operation, which denoises the entire frame before the frame is completed.

### 3.9 Rendering

#### 3.9.1 Asynchronous Rendering

Rendering is by default asynchronous (non-blocking), and is done by combining a frame buffer, renderer, camera, and world.

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.
To start an render task, use

```c
OSPFuture ospRenderFrame(OSPFrameBuffer, OSPRenderer, OSPCamera, OSPWorld);
```

This returns an OSPFuture handle, which can be used to synchronize with the application, cancel, or query for progress of the running task. When `ospRenderFrame` is called, there is no guarantee when the associated task will begin execution.

Progress of a running frame can be queried with the following API function

```c
float ospGetProgress(OSPFuture);
```

This returns the progress of the task in [0-1]. Applications can wait on the result of an asynchronous operation, or choose to only synchronize with a specific event. To synchronize with an OSPFuture use

```c
void ospWait(OSPFuture, OSPSyncEvent = OSP_TASK_FINISHED);
```

The following are values which can be synchronized with the application

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSP_NONE_FINISHED</td>
<td>Don’t wait for anything to be finished (immediately return from <code>ospWait</code>)</td>
</tr>
<tr>
<td>OSP_WORLD_COMMITED</td>
<td>Wait for the world to be committed (not yet implemented)</td>
</tr>
<tr>
<td>OSP_WORLD_RENDERED</td>
<td>Wait for the world to be rendered, but not post-processing operations</td>
</tr>
<tr>
<td></td>
<td>(Pixel/Tile/Frame Op)</td>
</tr>
<tr>
<td>OSP_FRAME_FINISHED</td>
<td>Wait for all rendering operations to complete</td>
</tr>
<tr>
<td>OSP_TASK_FINISHED</td>
<td>Wait on full completion of the task associated with the future. The underlying task</td>
</tr>
<tr>
<td></td>
<td>may involve one or more of the above synchronization events</td>
</tr>
</tbody>
</table>

Currently only rendering can be invoked asynchronously. However, future releases of OSPRay may add more asynchronous versions of API calls (and thus return OSPFuture).

Applications can query whether particular events are complete with

```c
int ospIsReady(OSPFuture, OSPSyncEvent = OSP_TASK_FINISHED);
```

As the given running task runs (as tracked by the OSPFuture), applications can query a boolean [0,1] result if the passed event has been completed.

### 3.9.2 Asynchronously Rendering and ospCommit()

The use of either `ospRenderFrame` or `ospRenderFrame` requires that all objects in the scene being rendered have been committed before rendering occurs. If a call to `ospCommit()` happens while a frame is rendered, the result is undefined behavior and should be avoided.

### 3.9.3 Synchronous Rendering

For convenience in certain use cases, ospray_util.h provides a synchronous version of `ospRenderFrame`:

```c
float ospRenderFrameBlocking(OSPFrameBuffer, OSPRenderer, OSPCamera, OSPWorld);
```

This version is the equivalent of:
ospRenderFrame
ospWait(f, OSP_TASK_FINISHED)
return ospGetVariance(fb)

This version is closest to ospRenderFrame from OSPRay v1.x.

### 3.10 Distributed rendering with MPI

The OSPRay MPI module is now a stand alone repository. It can be found on GitHub [here](https://github.com), where all code and documentation can be found.
Chapter 4
Examples

4.1 Tutorial

A minimal working example demonstrating how to use OSPRay can be found at apps/tutorials/ospTutorial.c.

An example of building ospTutorial.c with CMake can be found in apps/tutorials/ospTutorialFindospray/.

To build the tutorial on Linux, build it in a build directory with

gcc -std=c99 ../apps/ospTutorial/ospTutorial.c \
-I ../ospray/include -L . -lospray -Wl,-rpath,. -o ospTutorial

On Windows build it can be build manually in a “build_directory\$Configuration” directory with

c1 ..\..\apps\ospTutorial\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 accumulated-Frames.ppm.

Figure 4.1 – First frame.
4.2 ospExamples

Apart from tutorials, OSPRay comes with a C++ app called ospExamples which is an elaborate easy-to-use tutorial, with a single interface to try various OSPRay features. It is aimed at providing users with multiple simple scenes composed of basic geometry types, lights, volumes etc. to get started with OSPRay quickly.

ospExamples app runs a GLFWOSPRayWindow instance that manages instances of the camera, framebuffer, renderer and other OSPRay objects necessary to render an interactive scene. The scene is rendered on a GLFW window with an imgui GUI controls panel for the user to manipulate the scene at runtime.

The application is located in apps/ospExamples/ directory and can be built with CMake. It can be run from the build directory via:

./ospExamples <command-line-parameter>

The command line parameter is optional however.

4.2.1 Scenes

Different scenes can be selected from the scenes dropdown and each scene corresponds to an instance of a special detail::Builder struct. Example builders are located in apps/common/ospray_testing/builders/. These builders provide a usage guide for the OSPRay scene hierarchy and OSPRay API in the form of cpp wrappers. They instantiate and manage objects for the specific scene like cpp::Geometry, cpp::Volume, cpp::Light etc.
The `detail::Builder` base struct is mostly responsible for setting up OS-PRay world and objects common in all scenes (for eg: lighting and ground plane), which can be conveniently overridden in the derived builders.

Given below are different scenes listed with their string identifiers:

- **boxes** A simple scene with box geometry type.
- **cornell_box** A scene depicting a classic cornell box with quad mesh geometry type for rendering two cubes and a quad light type.
- **curves** A simple scene with curve geometry type and options to change curve-Basis. For details on different basis’ please check documentation of curves.
- **gravity_spheres_volume** A scene with structuredRegular type of volume.
- **gravity_spheres_isosurface** A scene depicting iso-surface rendering of gravity_spheres_volume using geometry type isosurface.
- **perlin_noise_volumes** An example scene with structuredRegular volume type depicting perlin noise.
- **random_spheres** A simple scene depicting sphere geometry type.
- **streamlines** A scene showcasing streamlines geometry derived from curve geometry type.
- **subdivision_cube** A scene with a cube of subdivision geometry type to showcase subdivision surfaces.
- **unstructured_volume** A simple scene with a volume of unstructured volume type.

### 4.2.2 Renderer

This app comes with three renderer options: `scivis`, `pathtracer` and `debug`. The app provides some common rendering controls like `pixel samples` and other more specific to the renderer type like `aoIntensity` for `scivis` renderer.