Prof. Dr. Philipp Slusallek
Dr. Pascal Grittmann
Teaching assistant: Lukas Auer

UNIVERSITÄT DES SAARLANDES

Computer Graphics
The Handbook

This handbook is your companion for working with *lightwave*, the ray tracing framework you will extend throughout the practical assignments. It is split into three parts:

- **Part I – Setup** is where you start. It walks you through installing the required tools, setting up your repository, and rendering your first image. Work through it step by step before doing anything else.
- **Part II – Basics** should be read once, right after you have finished the setup and before you start Sheet 1. It explains how the framework is organized, how scene files work, how to run the tests, and how to get started with debugging.
- **Part III – Reference** is meant to be consulted throughout the semester. It describes the scene format in detail and documents the tools lightwave offers you: the logger, arbitrary output variables (AOVs), the profiler, assertions, and more. You do not need to read it front to back; come back whenever you are looking for something specific.

The handbook never tells you *how* to solve an assignment – the sheets do that. Its goal is that you know your way around and do not miss a feature that could save you hours.

Contents

1 Setting up your workspace 2
1.1 Installing the tools 2
1.2 Setting up the repository 4
1.3 Visual Studio Code 5
1.4 Continuous integration 6
2 Rendering your first scene 6
2.1 Building and running lightwave 6
2.2 A first scene file 6
2.3 A dummy camera 7
3 Where to go from here 8
4 Lightwave Overview 9
4.1 Motivation 9
4.2 Codebase structure 9
4.3 Plugins and the registry 10
4.4 The main rendering loop 10
5 Running lightwave 11
5.1 Rendering a scene 11
5.2 Debug and Release builds 11
5.3 Reading the console output 11
6 Scene files 12
7 Tests 13
7.1 Running the tests 13
7.2 Integration tests 13
7.3 Unit tests 13
8 Debugging essentials 14
9 Good habits 14
10 Scene file reference 16
10.1 Nodes and parameters 16

10.2 Scene structure ... 17
10.3 Transforms ... 17
10.4 References and multiple jobs ... 17
10.5 Variables ... 18
10.6 Includes and file paths ... 18
10.7 Images ... 18
10.8 Postprocesses ... 19
10.9 Tests ... 19
11 Writing your own plugin ... 19
12 The logger ... 21
13 Arbitrary output variables (AOVs) ... 21
13.1 Usage ... 21
13.2 Semantics ... 22
13.3 Rules and limits ... 22
14 The profiler ... 22
15 Assertions ... 23
16 Debugging in depth ... 23
16.1 Hunting down a single pixel ... 23
16.2 The random number generator ... 24
16.3 Multi-threading pitfalls ... 24
16.4 Sanitizers ... 24
17 Framework internals ... 25
17.1 The render loop in detail ... 25
17.2 The math library ... 25
17.3 Code documentation ... 25
18 Writing your own tests ... 25
18.1 Integration tests ... 25
18.2 Unit tests ... 26
19 The Blender exporter ... 26

Part I – Setup

1 Setting up your workspace

> **Tip:** If you have trouble getting the framework to run on your machine, the best place to ask for help is the office hour.

1.1 Installing the tools

You will need:

- A C++ compiler
- git
- CMake
- python3 (to run the tests)
- An IDE or code editor
- An image viewer for `.exr` images (We recommend tev: https://github.com/Tom94/tev/)

Below you can find instructions for the recommended setup on Linux, macOS, and Windows.

1.1.1 Linux

Installing build tools and libraries

You can get a C++ compiler, git, and CMake from your package manager. E.g., on **Ubuntu**

```
sudo apt install -y build-essential git cmake
```

On **Fedora**

```
sudo dnf group install c-development
sudo dnf install git cmake
```

Installing tev

Download `tev.appimage` from https://github.com/Tom94/tev/releases/latest and put it in a location of your choice.

Installing Visual Studio Code

`.deb` and `.rpm` packages are available for download on https://code.visualstudio.com/

Refer to your package manager's documentation for how to install those; on some distros you can simply double click the downloaded file.

On **Ubuntu**, you may want to install Visual Studio Code via Snap:

```
sudo snap install --classic code
```

1.1.2 Windows

Installing build tools

Download and install Visual Studio (https://visualstudio.microsoft.com/). In the installation wizard, make sure to select *"C++ for Desktop"*, *"Git for Windows"* and *"Clang"*.

Installing tev

Go to https://github.com/Tom94/tev/releases and download the latest `tev.exe`. You may be asked to confirm that you trust the downloaded file, but it is safe to do so.

Installing Visual Studio Code

Go to https://code.visualstudio.com and download and run the installer.

1.1.3 macOS

Install the *Homebrew* package manager: Head over to https://brew.sh and copy the installation command into a terminal window. **Important:** There are additional commands (two `echo` and one `eval` command) that you will need to execute before being able to use Homebrew. These are listed in the *"Next steps"* section of the output of the installation command.

Installing build tools and libraries

Since Homebrew installs the *Xcode Command Line Tools* by default, our `C++` compiler and `git` are already set up for us. All that remains to be installed are the following packages:

```
brew install cmake # install CMake
```

Installing tev

Go to https://github.com/Tom94/tev/releases and download the latest `tev.dmg`. Open the `dmg` file and move tev to your Applications folder. For security reasons, the first time you run tev you must right-click on the application and click *"Open"*.

Installing Visual Studio Code This can be conveniently installed through homebrew:

```
brew install visual-studio-code
```

1.2 Setting up the repository

Each student group has access to two repositories: the *upstream repository* and the *group repository*.

a) The *upstream repository* is shared among all groups and is continuously updated throughout the semester with new assignments and possibly bug fixes. It contains the framework for our CG ray tracer, with placeholders that will be filled out through the practical assignments.

b) Your individual *group repository* is only visible to members of your group. This is where you will be implementing the practical assignments. You will get access to this repository after the deadline for team creation.

Adding the upstream repository

To start off, create a local git repository on your machine by navigating into an empty folder and using the `git init -b main` command. Once this repository is set up, you will need to add the *upstream repository* as a remote to it. This will allow you to check out new assignments once they are published. For this, you will need to open a terminal in Visual Studio Code and type the following commands:

```
git remote add upstream https://gitlab.cs.uni-saarland.de/cg1/ws26/upstream.git
```

Fetching new assignments

You can merge new assignments into your repository by running:

```
git fetch upstream
git merge upstream/main
```

Adding your group repository

Once your *group repository* is released, you want to add it as an additional remote that you can push your solutions to:

```
git remote add origin https://gitlab.cs.uni-saarland.de/cg1/ws26/group-a.git
```

The changes you made locally can then be pushed using:

```
git push --set-upstream origin main
```

At this step, you will probably be asked to authenticate yourself using your `GitLab` credentials[1]. After that, the initial push is complete and your changes should be visible on your projects `GitLab` page in the browser.

`GitLab` provides many tools for making collaborative work on a project easy. For example, you can create issues for exercises and assign them across team members.

[1] If you wish to clone your repository more securely using *SSH*, `GitLab` provides documentation on setting up the required keys

1.3 Visual Studio Code

Installing extensions

Before you can get started, you will first need to install two extensions:

a) The *"CMake Tools"* extension to be able to build your project.

b) The *"C/C++"* extension enables intelligent code completion and debugging.

Running your code

Once the extensions are set up, you should see two buttons in the bottom blue region of your Visual Studio Code window when opening your group repository:

a) A bug icon (*"Launch the debugger..."*). Use this to build and debug your ray tracer. While your code will run slower in debug mode, this is the best way to catch bugs and ensure everything is working as it should. You can set breakpoints in your code to pause execution and examine variables and function stacks, which are powerful tools to diagnose any issues your code might have.

b) A play icon (*"Launch the selected target..."*). Use this to build and execute your ray tracer once you are confident that everything is working as intended. Your code will run faster than in debug mode, but diagnosing crashes is not possible. You might want to test your code on low-resolution images in debug mode, and once that is working, render high-resolution images in this mode.

Passing a scene file to the debugger

To use the debugger or launch your renderer over the Visual Studio Code UI, you'll have to add a `settings.json` to your workspace (note that paths are relative to the executable here):

```
{
    "cmake.debugConfig": {
        "args": [
            "../scenes/my_first_scene.xml",
        ]
    }
}
```

To open and edit this file, press Windows+P (or Cmd+P on MacOS) and search for `>Preferences: Open Workspace Settings (JSON)`, this will automatically create the file if it doesn't exist yet.

Clang Format

To avoid inconsistent code formatting between the upstream and your own repository, you should make sure Visual Studio Code's formatter uses the provided `.clang-format` at the root of the project to automatically format on save. The *"C/C++"* extension offers a formatter that will look for the `.clang-format` file, however, you also need to make sure `Editor: Format On Save` is set to True in the user preferences of Visual Studio Code.

Committing changes

Visual Studio Code provides an intuitive user interface for `git`. This can be accessed through the *"Source Control"* tab in the sidebar. Here you can create commits using the *"Commit"* button, which you can push to `GitLab` using the *"Sync Changes"* button.

Using more powerful IDEs

Since our framework uses `cmake` as a build system, it is pretty easy to use other IDEs if you fancy something more powerful than Visual Studio Code. Just create a new `build` directory in your group repository and invoke `cmake` with the IDE of your choice. For example, for *Xcode*, you would run:

```
mkdir build && cd build
cmake -G Xcode ..
```

1.4 Continuous integration

Viewing the status of your builds

When you push your changes to `GitLab`, our CI will automatically build and execute your code on our servers. You can access the results of these runs by visiting your group repository's project page on `GitLab`. There you will find a *"CI/CD"* button (rocket icon) in the sidebar, which will take you to the *"Pipelines"* page. Here you can see the results for all pushes that you have made. A green *"passed"* badge indicates that your code compiled and ran on our servers successfully, while a red *"failed"* badge indicates that there were issues.

Diagnosing a *"failed"* build

You can view the logs to learn more about the issues that caused your code to fail on our servers. For this, click the red *"failed"* badge on your pipelines page, and then click the job that failed (indicated by a red cross mark next to it). Typically, the compiler will indicate to you what error and in which line it found it, which you can use to fix your code and try again. If the problem is not caused by compilation issues, feel free to contact your tutor or the teaching assistant.

2 Rendering your first scene

With everything installed, it is time to build lightwave and render an image. Do not worry if not every detail makes sense yet – Part II explains all of it.

2.1 Building and running lightwave

Build the program, either via the Visual Studio Code user interface or via the command line (`cmake -B build` followed by `cmake --build build`). You should then find the executable (named with intent `unnamed`) in the `build` directory. Running it without any arguments runs the unit tests:

```
./build/unnamed
```

You will see a failing unit test for now, since you have not implemented anything yet. That is expected (see Section 7.3).

2.2 A first scene file

Scenes are described in XML files. The following scene contains a single camera, no 3D objects, and a special integrator of type `camera` that visualizes the rays generated by the camera. Create a folder `scenes/` at the root of your repository (git ignores it) and save the scene as `scenes/my_first_scene.xml`:

```
<integrator type="camera">
    <scene>
        <camera type="perspective" id="camera">
            <integer name="width" value="400"/>
            <integer name="height" value="300"/>

            <string name="fovAxis" value="x"/>
            <float name="fov" value="90"/>

            <transform>
                <rotate axis="0,0,1" angle="30"/>
                <rotate axis="1,0,0" angle="30"/>
            </transform>
        </camera>
    </scene>
    <image id="my_first_scene"/>
    <sampler type="independent" count="16"/>
</integrator>
```

Render it by passing the path to the scene as the program's argument:

```
./build/unnamed scenes/my_first_scene.xml
```

2.3 A dummy camera

Running your program, you should get a `NOT_IMPLEMENTED` error originating from `src/cameras/perspective.cpp`. And indeed, we haven't implemented our camera yet. For now let's just write a dummy camera that generates a ray with coordinate xy equal to the pixel's normalized x coordinate:

```
CameraSample sample(const Point2 &normalized, Sampler &rng) const override
{
    return CameraSample{
                .ray = Ray(Vector(normalized.x(), normalized.x(), 0.f),
                           Vector(0.f, 0.f, 1.f)),
                .weight = Color(1.0f)};
}
```

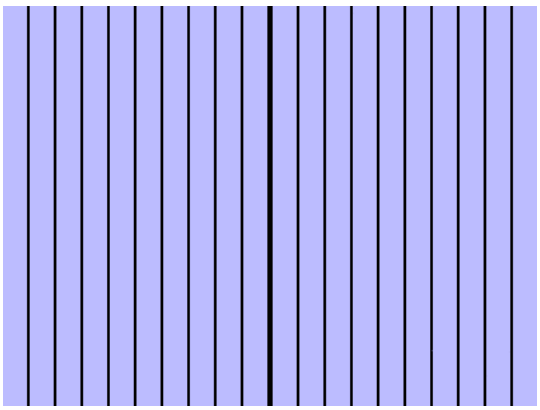

Figure 1: *The output image when rendering* `my_first_scene.xml`.

Make sure you also remove the `NOT_IMPLEMENTED` call in the camera constructor, compile the program and run it again. This time no error should be returned, only a few warnings about unused parameters in our scene file. This is expected, but where is our rendered image? The `id` of the `<image>` defines the name of the output `.exr` file, which is saved next to the scene file. You can either open `scenes/my_first_scene.exr` in tev or,

even better, make sure tev is running *before* rendering the scene. That way, rerunning the program with tev already open directly streams the image to tev's display and shows the image in Figure 1.

3 Where to go from here

Congratulations – your setup is complete! You have built lightwave, rendered your first image, and viewed it in tev.

Keep on reading. Before you start with Sheet 1, read Part II of this handbook. It explains what you just did: how the framework is organized, how scene files work, how the tests work, and how to debug your code. It will save you a lot of time later.

Then move on to Sheet 1. Its first exercise picks up exactly where you left off: the dummy camera you just wrote will be replaced by a proper perspective camera. After that, you will add 3D objects to your scene, transformations to move them around, and an integrator that visualizes their surface normals.

We wish you good luck and a lot of fun implementing your ray tracer!

> **Tip:** If you have enhanced our framework in a way that you think other groups might benefit from (e.g., through bug fixes or by adding support for more file formats), you are very welcome to create a pull request with your changes on the upstream repository. However, please make sure that your pull request does not contain solutions to any of our assignments.

Part II – Basics

4 Lightwave Overview

4.1 Motivation

Lightwave provides all the boilerplate a renderer needs but that does not teach you much about rendering: parsing scene files, loading meshes and images, saving and displaying images, multi-threading, a math library, and a test suite. What it does *not* provide is the rendering itself: cameras, shape intersections, materials, lights, and integrators are yours to write.

With every new assignment sheet, you merge new code from the upstream repository. This usually adds *scaffolds*: files with the right class structure, where the interesting parts are replaced by `NOT_IMPLEMENTED`. When your renderer reaches such a line, it stops with an error message that tells you the exact file and line, e.g.:

```
[error] src/cameras/perspective.cpp:31 NOT_IMPLEMENTED
```

You are free to change any line of the framework, including its interfaces. The only hard requirement is that your renderer can still read our test scenes and produces the right images for them.

4.2 Codebase structure

Path	Contents
`include/lightwave/`	The headers. They define the *interfaces* of all components (e.g., `shape.hpp`, `bsdf.hpp`, `integrator.hpp`) and the utilities (`math.hpp`, `color.hpp`, `logger.hpp`, …). Start here if you want to know what a component must do. `include/lightwave.hpp` includes everything at once.
`src/core/`	The framework itself: the scene parser, the main render loop (`integrator.cpp`), image loading and saving, streaming to tev, the logger, and the program entry point (`main.cpp`).
`src/cameras/`, `src/shapes/`, `src/integrators/`, …	The components (*plugins*) of your renderer, one folder per category: `bsdfs`, `cameras`, `emissions`, `integrators`, `lights`, `postprocesses`, `samplers`, `shapes`, `textures`, and `tests`. This is where most of your work happens. The BVH lives in `src/shapes/accel.hpp`.
`tests/`	The test scenes (`practical_N/*.xml`) together with the reference images (`*_ref.exr`), plus shared meshes and textures.
`unittests/`	Unit tests that check individual functions of your code (see Section 7).

Path	Contents
`run_tests.py`	The script that builds your renderer and runs all tests.
`features/`	The place to put renders of extra features for the rendering competition.
`deps/`, `cmake/`	External libraries and build system helpers. You will rarely need to touch these.

4.3 Plugins and the registry

Every component that can appear in a scene file is a *plugin*: a C++ class that derives from one of the interfaces and is registered under a name. For example, the camera integrator in `src/integrators/camera.cpp` ends with

```
REGISTER_INTEGRATOR(CameraIntegrator, "camera")
```

which tells lightwave to construct a `CameraIntegrator` whenever a scene file contains `<integrator type="camera">`. The tag name (`integrator`) selects the *category*, and the `type` attribute selects the plugin within that category. If you forget to register a class, or its file is not compiled, you will see an error such as

```
[error] <shape /> with unknown type="sphere"
```

All `.cpp` files below `src/` are compiled automatically; there is no list of source files you need to maintain. (If a newly created file is not picked up, re-run the CMake configure step.) The available categories are:

Tag	Base class	Header	Registration macro
`camera`	`Camera`	`camera.hpp`	`REGISTER_CAMERA`
`shape`	`Shape`	`shape.hpp`	`REGISTER_SHAPE`
`integrator`	`SamplingIntegrator`	`integrator.hpp`	`REGISTER_INTEGRATOR`
`bsdf`	`Bsdf`	`bsdf.hpp`	`REGISTER_BSDF`
`emission`	`Emission`	`emission.hpp`	`REGISTER_EMISSION`
`texture`	`Texture`	`texture.hpp`	`REGISTER_TEXTURE`
`light`	`Light`	`light.hpp`	`REGISTER_LIGHT`
`sampler`	`Sampler`	`sampler.hpp`	`REGISTER_SAMPLER`
`postprocess`	`Postprocess`	`postprocess.hpp`	`REGISTER_POSTPROCESS`
`test`	`Test`	`test.hpp`	`REGISTER_TEST`

A few tags such as `<scene>`, `<instance>`, `<material>`, `<image>`, and `<transform>` are provided by the framework and need no `type` attribute. Section 11 shows how to write a plugin from scratch.

4.4 The main rendering loop

It helps to know what happens when you run `./build/unnamed scene.xml`[2]:

a) `main.cpp` hands the file to the **scene parser**. For every tag, it collects the attributes and children into a `Properties` object and asks the **registry** to construct the matching plugin. The constructors of your

[2] Of course, your executable will have a better name than `unnamed` once you have done the corresponding task of Sheet 1.

plugins read their parameters from these properties. Loading runs in parallel, so several constructors may run at the same time.

b) Objects at the top level of the file that are **executable** – integrators, postprocesses, and tests – are executed one after another, in the order they appear in the file.

c) A `SamplingIntegrator` runs the **render loop** (`src/core/integrator.cpp`): the image is split into blocks of 64×64 pixels that are processed in parallel. For every pixel and every sample, the camera generates a ray and your integrator's `Li` function computes its color. The results are averaged per pixel.

d) While rendering, finished blocks are **streamed to tev** (if it is running). At the end, the image is **saved as an EXR file** next to the scene file.

5 Running lightwave

5.1 Rendering a scene

```
./build/unnamed                                  # no scene given: run the unit tests
./build/unnamed path/to/scene.xml                # render one scene
./build/unnamed a.xml b.xml                      # render several scenes in a row
./build/unnamed -Dspp=4 path/to/scene.xml        # set a scene variable
./build/unnamed tests/practical_1/aov_rectangle.xml   # run the test aov_rectangle
```

Scene variables are explained in Section 10.5. The output image is written next to the scene file and named after the `id` of its `<image>` node (e.g., `<image id="cbox"/>` produces `cbox.exr`).

Start tev before you render. Lightwave connects to tev on your machine (`127.0.0.1`, port `14158`, which is tev's default) and streams the image live while it is being rendered. If tev is not running, streaming is silently skipped and you only get the file.

5.2 Debug and Release builds

CMake can build your renderer in different configurations. In VS Code, you choose it in the status bar (CMake Tools extension); on the command line with `-DCMAKE_BUILD_TYPE=...`.

Configuration	Use it for
`Debug`	Stepping through code with the debugger. Variables are not optimized away, but rendering is *much* slower. Lightwave prints a warning at startup to remind you.
`RelWithDebInfo`	A good compromise: optimized, but with debug symbols, so crashes still show meaningful stack traces.
`Release`	Running the tests, rendering final images, and measuring performance.

> **Careful:** `run_tests.py` uses whatever configuration your `build/` folder currently has. If you forgot to switch back from `Debug`, the tests will be very slow and may even time out.

5.3 Reading the console output

Lightwave prints a lot of useful information. Take the time to read it:

- `[info]` lines tell you what is happening (loading, rendering progress, saving).
- `[warn]` lines deserve attention. In particular, lightwave warns you about attributes and children in the scene file that were **never queried** by the plugin:

```
[warn] attribute "fvo" was specified, but never queried
```

This catches typos in scene files, but also reminds you when your plugin does not read a parameter yet.

- `[error]` lines explain what went wrong. Errors that happen while loading a scene come with a chain of context, including the location in the scene file:

```
[error] src/core/parser.cpp:143 : defined in tests/practical_1/sphere_inside.xml:14:13
[error]   src/core/registry.cpp:50 : <shape /> with unknown type="sphere"
```

6 Scene files

Scenes are described in a simple XML format. Here is a complete scene that renders a sphere with the normal-visualizing integrator you will write in Sheet 1:

```xml
<integrator type="aov" variable="normals">
    <scene>
        <camera type="perspective">
            <integer name="width" value="640"/>
            <integer name="height" value="480"/>
            <float name="fov" value="40"/>
            <string name="fovAxis" value="x"/>
            <transform>
                <translate z="-5"/>
            </transform>
        </camera>

        <instance>
            <shape type="sphere"/>
            <transform>
                <scale value="0.5"/>
                <translate y="0.5"/>
            </transform>
        </instance>
    </scene>
    <sampler type="independent" count="16"/>
    <image id="my_sphere"/>
</integrator>
```

The rules are few:

- **The tag selects the category, `type` selects the plugin** (see Section 4.3).
- **Nesting defines relationships.** The integrator has a scene, a sampler, and an output image as children; the scene contains a camera and instances; an instance combines a shape with an optional material and transform.
- **Parameters** are given as `<float name="..." value="..."/>` (also `integer`, `boolean`, `string`, `color`, `vector`), or more compactly as attributes: `variable="normals"` and `count="16"` above are parameters, too.
- **Transforms** consist of `translate`, `scale`, `rotate`, `lookat`, and `matrix` operations, which are applied in the order they are written.

The scene files in `tests/` are the best examples you can find, since they use exactly the features you implement. The full format, including references, variables, includes, and postprocessing, is described in Section 10.

7 Tests

7.1 Running the tests

```
python3 run_tests.py                            # build, then run everything
python3 run_tests.py practical_1                # only the tests of one sheet
python3 run_tests.py practical_1/sphere_inside  # a single test
python3 run_tests.py "practical_2/pathtracing_*"  # all tests matching a pattern
python3 run_tests.py --skip-unit-tests          # only the integration tests
python3 run_tests.py --disable-build            # do not rebuild first
```

The script first builds your renderer (`cmake --build build`), so it needs a configured `build/` folder. It then runs two kinds of tests:

- **Unit tests** live in `unittests/` and check individual functions of your code in isolation (see Section 7.3).
- **Integration tests** are the `tests/practical_N/*.xml` scenes. Each one renders a scene with your renderer and compares the result against a reference image.

7.2 Integration tests

A test scene wraps a normal render job in a `<test>` node:

```
<test type="image" id="aov_rectangle">
    <integrator type="aov" variable="normals">
        ...
    </integrator>
</test>
```

It renders the image into `aov_rectangle_test.exr` and compares it with `aov_rectangle_ref.exr` from the same folder. The test fails if

- the image contains a NaN, infinite, or (if disallowed) negative value – the error tells you the pixel,
- the mean absolute error (MAE) exceeds a threshold – this catches images that are wrong *somewhere*,
- the mean error (ME) exceeds a threshold – this catches images that are too bright or too dark *on average*, which is a sign of bias.

If tev is running, both images are shown there. Select your `_test` image and *Shift+Click* the `_ref` image to see the difference between them. This usually tells you much more than the error number.

> **Tip:** Our tests use simple error metrics and are only meant to guide you. A passing test does not guarantee that your implementation is correct, and a failing test with an error just above the threshold is worth a closer look in tev.

The same tests run on our GitLab CI for every push (see Section 1.4). Passing them there is what counts.

7.3 Unit tests

Unit tests check single functions of your plugins without rendering an image, e.g., whether your sphere reports the correct bounding box or whether a BSDF returns a finite value. They are written with the Catch2 framework and live in `unittests/practical_N/` , with one file per component. They are compiled into your renderer, so there are two ways to run them:

```
python3 run_tests.py     # runs the unit tests first, then the integration tests
./build/unnamed          # no scene given: runs only the unit tests
```

For every failing test, `run_tests.py` prints the file, the line, and the check that failed. Running `./build/unnamed` directly shows the full Catch2 output, including the values that were compared. Since unit tests look at one function at a time, they are a quick way to locate a bug before you look at any image. Keep in mind:

- **All unit tests always run.** Selecting tests (e.g., `run_tests.py practical_1`) only affects the integration tests. Tests for components you have not implemented yet will fail, so focus on the ones in the `unittests/practical_N/` folder of your current sheet.
- **A `NOT_IMPLEMENTED` stops the whole unit test run.** The test that reaches it is reported as failed, and all remaining tests are skipped. This is normal while you are still working through the scaffolds. Use `--skip-unit-tests` to focus on the integration tests in the meantime.
- **Debugging a unit test:** start your renderer in the debugger without any scene argument (remove it from `cmake.debugConfig` in your `settings.json`, see Section 1.3) and set a breakpoint in the test or in your own code.
- **Passing unit tests are not the whole story.** They check individual parts of your code; only the integration tests tell you whether the parts produce the right image together.

You can also write unit tests of your own, see Section 18.

8 Debugging essentials

Most bugs in a renderer show up as a wrong image rather than a crash. These are the first tools to reach for:

a) **Look at the image.** Compare against the reference in tev, inspect pixel values by hovering over them, and adjust the exposure to see very dark or very bright regions.

b) **Visualize intermediate quantities.** Your AOV integrator from Sheet 1 shows normals; the `aov()` function (Section 13) lets *any* integrator write extra images, e.g., the contribution of each bounce.

c) **Print values** with the logger (Section 12), but only for a few pixels, not all of them.

d) **Check your assumptions** with assertions such as `assert_normalized(direction, {})` (Section 15). They are enabled in all builds and tell you the file, line, and offending value when something goes wrong.

e) **Use the debugger.** Build in `Debug`, set a breakpoint, and start the debugger from VS Code (see Section 1.3). Stepping through code is much easier with a single thread: uncomment `#define SINGLE_THREADED` in `include/lightwave/parallel.hpp` (only effective in `Debug` builds).

When a test *crashes* (`crashed due to segmentation fault`), run that scene directly from the debugger. It will stop exactly where the crash happens.

> **Tip:** Renders in lightwave are deterministic: every pixel and sample gets its own fixed random seed, independent of the number of threads. A bug that shows up at a certain pixel will show up there again in the next run, so you can go hunting for it (Section 16.1).

9 Good habits

- **Keep your own scenes in `scenes/`.** This folder is ignored by git, so it is a good place for experiments and large downloaded assets. Files you want to hand in (e.g., the Blender scene of Sheet 3) must go somewhere else.
- **EXR files are not committed**, except for reference images (`*_ref.exr`). Your test outputs will not clutter your repository.

- **Commit and push often.** The CI results tell you whether your code compiles and passes the tests on our machines, too.
- **Merge the upstream repository** when a new sheet is published (see Section 1.2), and read the new scaffolds before you start implementing.

Part III – Reference

10 Scene file reference

10.1 Nodes and parameters

Every XML tag is one of the following:

Tag	Meaning
`<integrator>`, `<shape>`, …	An **object** (plugin) of the given category. Most need a `type`. Objects may carry an `id` (see below) and, when used as a named parameter of their parent, a `name`.
`<float>`, `<integer>`, `<boolean>`, `<string>`, `<color>`, `<vector>`	A **parameter** of the parent object, with `name` and `value` attributes.
`<translate>`, `<scale>`, `<rotate>`, `<lookat>`, `<matrix>`	An operation inside a `<transform>`.
`<ref id="..."/>`	A **reference** to an object defined earlier.
`<include filename="..."/>`	Inserts another scene file at this position.
`<default name="..." value="..."/>`	Defines a default value for a **variable**.
`<lightwave>`	An optional wrapper around the whole file; it has no effect.

Value formats. Colors and vectors are given as comma-separated components (`"0.9,0.1,0.1"`); a single number (`"0.5"`) sets all components. Booleans are `true` or `false`. Angles are given in degrees.

Attributes as parameters. Every attribute of an object tag other than `type`, `id`, and `name` becomes a parameter. The following two lines are equivalent:

```
<sampler type="independent"><integer name="count" value="64"/></sampler>
<sampler type="independent" count="64"/>
```

Named child objects. Objects can be parameters too. The `name` attribute tells the parent which role the child plays; the plugin then queries it by that name:

```
<bsdf type="diffuse">
    <texture name="albedo" type="constant" value="0.8"/>
</bsdf>
```

10.2 Scene structure

A typical rendering job looks like this:

```
<integrator type="...">           <!-- how to render -->
    <scene>                       <!-- what to render -->
        <camera type="perspective"> ... </camera>
        <instance>                <!-- one object in the scene -->
            <shape type="..."/>
            <material>            <!-- optional: its appearance -->
                <bsdf type="..."> ... </bsdf>
                <emission type="..."> ... </emission>
            </material>
            <transform> ... </transform>   <!-- optional: its placement -->
        </instance>
        <light type="..."> ... </light>    <!-- optional: lights -->
    </scene>
    <sampler type="independent" count="64"/>
    <image id="output"/>
</integrator>
```

Which plugin types exist depends on how far you are in the course. The test scenes of each sheet show every type and parameter that your implementation needs to support.

10.3 Transforms

A `<transform>` places cameras and instances in the world. It consists of any number of operations that are **applied in the order they are written**:

```
<transform>
    <scale value="0.5"/>              <!-- uniform scaling -->
    <scale x="2" y="1" z="1"/>        <!-- non-uniform scaling -->
    <rotate axis="0,1,0" angle="45"/> <!-- angle in degrees -->
    <translate x="1" z="-3"/>         <!-- omitted components are 0 -->
</transform>
```

`<lookat origin="0,1,-5" target="0,0,0" up="0,1,0"/>` positions a camera at `origin` looking at `target`, and `<matrix value="..."/>` appends a full 4×4 matrix (16 values). For cameras, the transform maps from camera space to world space.

10.4 References and multiple jobs

Any object can be given an `id` and then be reused with `<ref id="..."/>`. The object must be defined *before* it is referenced. This avoids repeating the same material many times:

```
<material id="white">
    <bsdf type="diffuse"><texture name="albedo" type="constant" value="0.8"/></bsdf>
</material>

<instance> <shape type="rectangle"/> <ref id="white"/> </instance>
<instance> <shape type="sphere"/>    <ref id="white"/> </instance>
```

References also allow you to put **several jobs** into one file. Everything at the top level of the file is executed in order, so you can define a scene once and render it with different integrators:

```
<scene id="scene"> ... </scene>

<integrator type="aov" variable="normals">
    <ref id="scene"/>
    <sampler type="independent" count="16"/>
    <image id="normals"/>
</integrator>

<integrator type="pathtracer">
    <ref id="scene"/>
    <sampler type="independent" count="256"/>
    <image id="beauty"/>
</integrator>
```

A reference can also be given a `name`, e.g., `<ref name="input" id="beauty"/>` passes the object as the parameter `input` (this is how postprocesses receive their input, see Section 10.8).

10.5 Variables

Attribute values can contain variables in the form `${name}`. Their values come either from the command line or from a `<default>` in the scene file, where the command line takes precedence:

```
<default name="spp" value="256"/>
...
<sampler type="independent" count="${spp}"/>
```

```
./build/unnamed scene.xml            # renders with 256 spp
./build/unnamed -Dspp=4 scene.xml    # quick preview with 4 spp
```

This is handy for switching between quick previews and final renders, or for trying out different parameter values without editing the file. Using a variable that has no value is an error.

10.6 Includes and file paths

`<include filename="materials.xml"/>` inserts the contents of another file at this position. This is useful to share materials or geometry between several scenes.

All file paths (meshes, textures, includes) are **relative to the scene file that contains them**, not to your working directory. The same holds for output images, which are saved next to the scene file.

10.7 Images

The `<image>` tag serves two purposes:

- **As output**, `<image id="name"/>` is the image an integrator or postprocess writes into. It is saved as `name.exr`.
- **As input**, `<image filename="..."/>` loads an image from disk. EXR, HDR, PNG, JPG, and other common formats are supported. Non-HDR formats such as PNG and JPG are stored with a gamma curve; lightwave converts them to linear values when loading, unless you specify `linear="true"` (e.g., for data such as roughness maps that are already linear). With `channel="r"` (or `g`, `b`, `a`), only a single channel is loaded as a grayscale image.

Lightwave always saves EXR files, which store linear floating point values without clamping. That is why you should view them in tev rather than a normal image viewer.

> **Tip:** Every EXR file written by lightwave contains the complete console log of the render as metadata (attribute `log`). If you are wondering months later with which settings an image was rendered, the answer is in the file.

10.8 Postprocesses

A postprocess takes an existing image and produces a new one. Like integrators, postprocesses are placed at the top level and are executed in order. The input is passed as the named parameter `input`, the output is a child `<image>`:

```
<!-- postprocess an image from disk -->
<postprocess type="tonemap">
    <image name="input" filename="render.exr"/>
    <image id="render_tonemapped"/>
</postprocess>
```

To postprocess a render in the same file, give the integrator's output image an `id` and reference it:

```
<integrator type="pathtracer">
    <ref id="scene"/>
    <sampler type="independent" count="256"/>
    <image id="beauty"/>
</integrator>

<postprocess type="tonemap">
    <ref name="input" id="beauty"/>
    <image id="beauty_tonemapped"/>
</postprocess>
```

Postprocesses can be chained: the output of one can be the input of the next. If you implement your own (e.g., for the rendering competition), derive from `Postprocess` (see `include/lightwave/postprocess.hpp`): the constructor already provides `m_input` and `m_output`, and your `execute()` fills `m_output` and saves it.

10.9 Tests

`<test type="image" id="...">` wraps exactly one executable (an integrator or postprocess). Its parameters are:

Parameter	Default	Meaning
`mae`	`0.1`	Maximum allowed mean absolute error.
`me`	`0.0002`	Maximum allowed (absolute value of the) mean error.
`allowNegative`	`true`	Whether negative pixel values are accepted.

See Section 18 for how to write your own tests.

11 Writing your own plugin

Adding a new plugin requires one `.cpp` file in the right folder. Here is the skeleton of an integrator with a few parameters:

```cpp
#include <lightwave.hpp>

namespace lightwave {

class MyIntegrator : public SamplingIntegrator {
    int m_depth;
    Color m_tint;
    enum class Mode { Fast, Accurate } m_mode;

public:
    MyIntegrator(const Properties &properties) : SamplingIntegrator(properties) {
        m_depth = properties.get<int>("depth", 3);   // optional, default 3
        m_tint  = properties.get<Color>("tint");     // required
        m_mode  = properties.getEnum<Mode>("mode", Mode::Fast, {
            { "fast",     Mode::Fast     },
            { "accurate", Mode::Accurate },
        });
    }

    Color Li(const Ray &ray, Sampler &rng) override {
        Intersection its = m_scene->intersect(ray, rng);
        ...
    }

    std::string toString() const override {
        return tfm::format("MyIntegrator[\n  depth = %d\n]", m_depth);
    }
};

} // namespace lightwave

REGISTER_INTEGRATOR(MyIntegrator, "mine")
```

The most important functions of `Properties` (see `include/lightwave/properties.hpp`) are:

Function	Returns
`get<T>("name")`	The parameter `name`; an error if it is missing.
`get<T>("name", fallback)`	The parameter, or `fallback` if it is missing.
`getEnum<T>("name", fallback, {...})`	Maps a string parameter to a value of your choice; an error for unknown strings.
`get<Texture>("name")`, `getOptional<Texture>("name")`	A named child object (e.g., `<texture name="albedo" .../>`).
`getChild<T>()`, `getOptionalChild<T>()`	The unnamed child of type `T` (an error if there are several).
`getChildren<T>()`	All unnamed children of type `T`.
`has("name")`	Whether the parameter is present.

Careful: Errors in the constructor should be reported with `lightwave_throw("message %s", value)`; you will get a nice error message pointing to the scene file. Do *not* throw exceptions in code that runs during rendering (`Li`, `intersect`, `evaluate`, ...): their mere presence slows down the renderer considerably. Use assertions (Section 15) instead.

12 The logger

`logger` is a global object that prints messages, and it is safe to use from multiple threads at the same time:

```
logger(EDebug, "only interesting while debugging");
logger(EInfo,  "loaded %d triangles in %.2f seconds", count, seconds);
logger(EWarn,  "roughness %f is out of range, clamping", roughness);
logger(EError, "something went badly wrong");
```

The format string works like `printf`, but `%s` accepts almost anything that can be printed: lightwave's `Vector`, `Point`, `Color`, `Ray`, `Frame`, `Bounds`, and matrices, as well as pointers to lightwave objects (which use their `toString()`; use `.get()` on a `ref<T>`):

```
logger(EInfo, "hit %s with normal %s", its.position, its.shadingNormal);
logger(EInfo, "the scene is %s", m_scene.get());
```

> **Careful:** Your `Li` function is called millions of times. Logging every call floods your terminal and slows rendering to a crawl. Log only when something interesting happens (e.g., a value is out of range), only for one pixel (Section 16.1), or render a tiny image with one sample per pixel.

For long-running work of your own (e.g., an expensive precomputation), the `ProgressReporter` from `logger.hpp` displays a progress bar with an estimate of the remaining time:

```
ProgressReporter progress{ "precompute", totalSteps };
for (...) { ...; progress += 1; }
progress.finish();
```

13 Arbitrary output variables (AOVs)

The name AOV is used for two different things in lightwave. The *AOV integrator* from Sheet 1 is a plugin you write: it renders one debug quantity (such as the normals) *instead of* the actual image. The **`aov()` function** described here is a framework feature that lets any code write *additional* images while the actual rendering continues as usual. It is one of the most useful debugging tools you have.

13.1 Usage

```
#include <lightwave/aov.hpp>   // already included by lightwave.hpp

Color Li(const Ray &ray, Sampler &rng) override {
    Intersection its = m_scene->intersect(ray, rng);
    aov("hit") = its ? Color(1) : Color(0);
    aov("uv")  = its ? Color(its.uv.x(), its.uv.y(), 0) : Color(0);
    ...
    aov("bounce %d", ray.depth) += contribution;
    ...
}
```

Each distinct name creates a full-resolution image. No declaration or setup is needed. All of them are:

- **streamed live to tev** as layers of the rendered image, so you can switch between them while rendering,
- **saved as layers of the output EXR file**, next to the actual image (the "beauty" image).

13.2 Semantics

- `aov(...)` returns a reference to a `Color` that belongs to the **current sample**, starting out black. Use `=` to set it and `+=` to accumulate into it within the sample.
- The final value of a pixel is the **mean over all samples** of that pixel. Samples that do not write to an AOV count as zero. In the example above, `hit` therefore shows the fraction of samples that hit an object – a soft mask with anti-aliased edges.
- The name works like a `printf` format string, which makes it easy to create families of AOVs (`"bounce %d"`).

13.3 Rules and limits

- The format string must be a **string literal**, and the arguments must be **numbers or enums** (no strings). The compiler enforces this.
- There are at most 64 AOVs per image; any further ones are merged into a single `overflow` layer with a warning. Never format a continuous value into the name (such as `aov("t=%f", its.t)`) – that creates a new image for every value.
- Calling `aov()` is cheap (a hash and a table lookup) but not free. `aovs.setEnabled(false)` turns all AOVs off, e.g., when you measure performance.
- Integrators that write to pixels other than the one being rendered (such as light tracers, which you might build for the rendering competition) can use the overload `aov(pixel, "name") += value`, which addresses a pixel explicitly.

14 The profiler

When your renderer is slow, the profiler tells you *where* the time goes. Mark a block of code with `PROFILE`; the measurement covers everything from that line until the end of the enclosing scope:

```
Color evaluate(...) const override {
    PROFILE("My BSDF")
    ...
}
```

When the program exits, lightwave prints a table of all profiled blocks, sorted by time. Nested blocks are indented below the block they were called from. An example (the numbers are made up):

```
block                    global    local   times called

Intersect                 58.2%    58.2%   2,097,152
  Rectangle               20.4%    35.1%   9,437,184
  Remainder               37.8%    64.9%
My BSDF                   14.9%    14.9%   1,310,720
Remainder                 26.9%    26.9%
```

- **global**: share of the total time of all threads.
- **local**: share of the time of the parent block.
- **times called**: how often the block was entered. Surprisingly high numbers are often a hint by themselves.
- **Remainder**: time spent in the parent that is not covered by any profiled child block.

The framework already profiles a few central functions, such as `Intersect` (`Scene::intersect`) and `Transmittance` (shadow rays), so you will see a table from the very beginning. Add your own blocks to narrow down bottlenecks, e.g., when you are working on the BVH.

> **Tip:** Always profile `Release` builds: a `Debug` build spends its time in very different places. The profiler itself adds a small overhead per block. If you want to get rid of it entirely, replace the `PROFILE` macro in `include/lightwave/profiler.hpp` by the empty definition that is commented out right below it.

15 Assertions

Lightwave provides assertions that check common assumptions and abort with a helpful message if they are violated. Unlike the standard `assert`, they are **also enabled in Release builds** (they are cheap enough), so they catch bugs during the tests as well.

Macro	Checks
`assert_condition(cond, ctx)`	An arbitrary condition.
`assert_finite(value, ctx)`	That a number, vector, or color contains no NaN or infinity.
`assert_normalized(v, ctx)`	That a vector has unit length.
`assert_orthogonal(a, b, ctx)`	That two vectors are orthogonal.

The second argument is a block of code that runs before aborting. Use it to log additional context:

```
assert_finite(weight, {
    logger(EError, "pdf = %f, cosine = %f", pdf, cosine);
});
assert_normalized(ray.direction, {});   // no additional context
```

A failed assertion prints the file, line, the violated condition, and the offending value:

```
[error] src/integrators/mine.cpp:42 : weight (= Color[nan, nan, nan]) is not finite
[error] pdf = 0.000000, cosine = 0.000000
```

> **Tip:** NaNs are among the most common and most frustrating bugs in a renderer: a single NaN spreads through every computation it touches and ends up as a black or missing pixel. When a test reports `nan encountered at pixel ...`, sprinkle `assert_finite` over the places where the value could come from, and the first assertion that fails tells you where the NaN was born.

16 Debugging in depth

16.1 Hunting down a single pixel

Since renders are deterministic, a problem that appears at a certain pixel will appear there in every run. Hover over the pixel in tev to read its coordinates. Then find the pixel in the render loop in `src/core/integrator.cpp`: this is where the camera ray for each `pixel` is generated and `Li` is called. Two ways to investigate:

- Add a line such as `if (pixel.x() == 123 && pixel.y() == 45) { ... }` just before `Li` is called and set a breakpoint inside it. From there you can step into your `Li` function. (Remember to remove the line afterwards.)
- To print values from deep inside your code for this pixel only, set a global flag there, e.g., a `thread_local bool` that is `true` only for the pixel of interest, and check it before logging.

16.2 The random number generator

The `independent` sampler seeds its random numbers from the pixel, the sample index, and a seed. The `seed` parameter changes the latter:

```
<sampler type="independent" count="64" seed="7"/>
```

This is useful to tell **noise** from **bias**. Noise changes from seed to seed and goes away as you increase the sample count. A difference to the reference that stays when you change the seed and increase the sample count is a systematic error in your code.

16.3 Multi-threading pitfalls

`Li` (and everything it calls) runs on many threads at the same time, and all threads share the same integrator, scene, and material objects. Therefore:

- **Do not modify member variables** during rendering. Use local variables instead. Member variables should only be written in constructors.
- Every rendering thread gets its own copy of the sampler, so always use the `Sampler &rng` that is passed to you.
- `logger` and `aov()` are thread-safe. For your own shared accumulations, `include/lightwave/parallel.hpp` provides `atomicAdd` for numbers and colors.
- To parallelize your own loops (e.g., in a postprocess or a precomputation), use `for_each_parallel` from the same header.
- If a bug disappears with `SINGLE_THREADED` defined, it is very likely a race condition.

16.4 Sanitizers

Sanitizers make the compiler insert checks into your program that detect certain classes of bugs the moment they happen. They are available with GCC and Clang (only the address sanitizer with MSVC) and are enabled through CMake options when configuring your project:

CMake option	Detects
`LW_USE_ADDRESS_SANITIZER`	Out-of-bounds accesses, use-after-free, and other memory errors.
`LW_USE_UNDEFINED_BEHAVIOR_SANITIZER`	Undefined behavior, e.g., signed integer overflow or invalid casts.
`LW_USE_THREAD_SANITIZER`	Race conditions between threads.

```
cmake -B build -DCMAKE_BUILD_TYPE=Debug -DLW_USE_ADDRESS_SANITIZER=ON
```

In VS Code, add the option to your workspace settings (`settings.json` , see Section 1.3) instead:

```
"cmake.configureSettings": { "LW_USE_ADDRESS_SANITIZER": "ON" }
```

Sanitizers only take effect in `Debug` and `RelWithDebInfo` builds and slow down your program considerably, so turn them off again when you are done. (CMake remembers options between runs, so you have to explicitly set them to `OFF` .)

17 Framework internals

17.1 The render loop in detail

`SamplingIntegrator::execute()` in `src/core/integrator.cpp` is short and worth reading. Some details:

- The image is processed in blocks of 64×64 pixels in a spiral order starting at the center, distributed over all CPU cores.
- For very large jobs (more than 10^8 samples in total, e.g., 1000×1000 pixels with 128 samples each), the image is rendered **progressively** in passes with growing sample counts. The file is saved after each pass, so you can abort a long render and still keep an image.
- If you need full control over rendering (e.g., for algorithms that do not fit the "one pixel at a time" pattern), your integrator can override `execute()` instead of implementing `Li`.

17.2 The math library

`include/lightwave/math.hpp` and `color.hpp` contain much more than you might expect. Before writing a helper function yourself, check whether it exists already:

What	Where / names
Vectors and points	`Vector`, `Point`, `Vector2`, `Point2`, `Point2i`, … with `dot`, `cross`, `length`, `normalized`, component access via `x()`, `y()`, `z()`. Points and vectors are different types on purpose.
Colors	`Color` with arithmetic, `luminance()`, `mean()`, `Color::black()`, `Color::white()`.
Matrices and transforms	`Matrix4x4`; `Transform` (`transform.hpp`) with `apply`, `inverse`, and `applyNormal`.
Bounding boxes	`Bounds` (3D) and `Bounds2i` (pixel ranges).
Constants	`Pi`, `InvPi`, `Inv2Pi`, `Inv4Pi`, `Deg2Rad`, `Epsilon`, `Infinity`, …
Safe math	`sqr`, `safe_sqrt`, `safe_acos`, `clamp`, `saturate` – use the `safe_` variants whenever rounding errors could push an argument out of range.
Sample warping	`include/lightwave/warp.hpp`: mapping uniform random numbers to disks, spheres, and hemispheres, together with their pdfs.

17.3 Code documentation

The headers are documented with Doxygen comments. Hover over a function in your IDE to read them, or generate browsable HTML documentation by running `doxygen` at the root of the repository (requires Doxygen); the result is in `docs/html/index.html`.

> **Tip:** On Linux and macOS, the build creates a `compile_commands.json` at the root of the repository. If you prefer `clangd` over the Microsoft C/C++ extension for code completion, it works out of the box.

18 Writing your own tests

18.1 Integration tests

You can wrap any of your own scenes in a `<test type="image" id="...">` node. A useful application is **regression testing**: before optimizing or refactoring code (e.g., the BVH), render a reference with the current

version, and check afterwards that the output has not changed. To create the reference image `<id>_ref.exr` instead of comparing against it, set the environment variable `reference`:

```
reference=1 ./build/unnamed scenes/my_test.xml   # Linux / macOS
```

In this mode, the `independent` sampler uses a different default seed, so that the noise in the reference is independent of the noise in the tested image – just like for our references.

> **Careful:** Never do this in the `tests/` folder: it overwrites our reference images, and you can no longer trust your test results. If it happens by accident, restore them with `git checkout tests/`.

18.2 Unit tests

Unit tests are written with the Catch2 framework. Every `.cpp` file in `unittests/` is compiled into your renderer automatically, so adding a test is as simple as creating a file:

```
#include <catch_amalgamated.hpp>
#include <lightwave.hpp>

namespace lightwave {

TEST_CASE("sphere samples are normalized", "[warp]") {
    for (float u : { 0.f, 0.3f, 0.7f, 0.99f }) {
        for (float v : { 0.f, 0.3f, 0.7f, 0.99f }) {
            const Vector w = squareToUniformSphere(Point2(u, v));
            REQUIRE_THAT(w.length(), Catch::Matchers::WithinAbs(1, 1e-5));
        }
    }
}

} // namespace lightwave
```

`REQUIRE` stops the test at the first failure, while `CHECK` records the failure and continues. Look at the existing files in `unittests/` for more examples, e.g., of testing a plugin that is constructed from `Properties`.

19 The Blender exporter

Blendwave is a blender extension that exports scenes from Blender to lightwave. After downloading the newest release from GitLab,you install it in Blender via *Edit* → *Preferences* → *Add-ons*. Export your scenes by pressing the `n` key in layout mode to open the sidebar and navigating the lightwave menu. Sheet 3 guides you through this step by step.

The exporter converts objects into triangle meshes (`.ply` files) and translates cameras, lights, and materials to the closest lightwave counterparts. Keep in mind:

- Exported scenes can only use plugins that your renderer implements. If a scene uses a type you have not implemented yet, you will get an `unknown type` error; replace it in the XML file or in Blender.
- Blender materials are only supported to a certain degree. The console output of lightwave (warnings about unqueried attributes) tells you which parameters were ignored.
- Exported scenes come with a `<sampler>` count and resolution that you may want to reduce for previews – or turn into a variable (Section 10.5).