About Lightmetrica Version 3[*]

Hisanari Otsu[†]

August 12, 2019

1 About This Article

What's Lightmetrica. *Lightmetrica*[1] is a research-oriented renderer that has been in development for years as my half-hobby project. The motivation of the project is to develop a practical research-oriented renderer and environment for those who wish to explore various new problems in rendering, such as a development of rendering techniques. The renderer is designed to focus on extensibility and verifiability in mind. By extensibility the developer can extend the renderer and implement a new feature. By venerability the developer can assure what is happening in the extended feature for instance being correct or not.

In rendering research we need to compare multiple approaches in a controlled configurations. The renderer shall be extensible to compare these approaches with the other part of the renderer being in the same configuration. In the same time, the developer do not want to concern unrelated part of the renderer. For instance, a developer who wants to extend acceleration structure should not know about materials. Typically, the progress of developing a new feature forms a sequence of trial-and-errors. It is thus also crucial for a renderer to be easy to verify what is happening inside, for instance, determining the observed phenomena is a bug or a systematic error in algorithms.

Short History. I started the development of Lightmetrica in 2014. The initial version was rather an experimental prototype. In 2015, I extended this prototype to the version 2 (I simply call this *Version 2*)[2]. Fortunately, the development of Version 2 could be affiliated by MITOU Program[3], a goverment-funded program by Ministry of Economy, Trade and Industry, Japan. The basic design of the renderer is fixed in this version. Successfully, I could use Version 2 in various research projects including several SIGGRAPH/SIGGRAPH Asia paper projects. In 2019, I decided to reimplement the whole renderer and I call this version as Version 3.

[*]This article is written for rtcamp7 advent calender.

[†]Twitter: @h2p_perim

[1]`https://github.com/lightmetrica/lightmetrica-v3`

[2]`https://github.com/hi2p-perim/lightmetrica-v2`

[3]`https://www.ipa.go.jp/english/humandev/third.html`

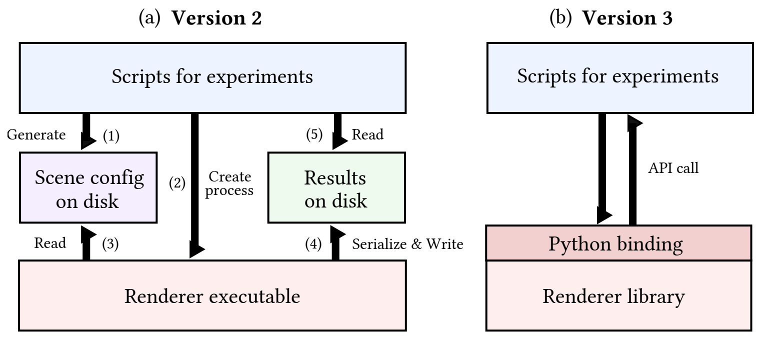


Figure 1: Use-case of the renderer in an experiments. In the research context, we often use scripting languages (e.g., Python) to orchestrate the experiments. (a) In Version 2, the renderer is provided as an independent application. To integrate the renderer into the process of the experiment, we thus needed an indirect steps to execute the renderer. The numbers (1) to (5) in the figure indicate the necessary operations that must be done in the experimental script. Although it can be alleviated by a wrapper script, it is tedious to maintain this workflow inside the experiments. (b) In Version 3, on the other hand, the renderer is provided as an library with Python binding. The user can directly call the library features from the experimental scripts.

Why We Need Version 3. Given the experiences and the accumulation of knowledge through various research projects, I have identified possible improvements and missing requirements or over-/under-specification of the design of Version 2. Some of them are identified and patched in the course of the projects on the top of the existing design, with suboptimal design choices that sacrifice the usability and performance of the framework. These codes are now becoming a technical debt through an accumulation of repetitive patches from each of research projects. Furthermore, some of the requirements are completely incompatible to the existing design of Version 2. In order to conduct a refactoring, at best, I could expect massive change to the existing design and implementation.

Therefore, I decided to overhaul the framework based on a completely new codebase rather than refactoring. I can safely say although the renderer still preserves the name and continuing version number, the internal design is completely different. Some of the design are inherited from Version 2 and some are not. In the the following sections we will look into the design choices that I have taken for Version 3, by mainly focusing on the difference between Version 2.

2 About Version 3: What's New?

2.1 Renderer as a Library

Version 2 – Renderer as an Independent Application. In Version 2, all features of the renderer must be accessed through an executable (Figure. 1.a). The parameters to the renderer must be passed by a scene definition file written in YAML format, and the outputs from the renderer must be written onto the disk. This execution model works great if the renderer used as an independent renderer. In a research project, however, a renderer is usually integrated into the workflow of experiments typically written in a scripting language, for instance to organize various input parameters and to analyze the outputs from the renderer. In my research, I like to organize the experiments with Python scripts.

After the experience of various research projects, I found a renderer as an independent executable is rather suboptimal in this workflow. First, due to the separated layers of renderer IO, the experimental code needs to generate the configuration on the fly. This requires the user to maintain a layer to serialize input parameters to the renderer as a specific scene configuration format, which needs to be managed throughout the experiments. In Version 2, I have used a parameterized scene definition file (using Jinja2 library[4]) and generated the scene definition file on the fly where the scene definition can contain arbitrary parameters to the extended features. A problem is managing this intermediate layer is error prone and it becomes hard to manage as the research progresses. Once we modify the parameters, we need to modify both of the scene file generation and experimental codes. In the early stage of the development, it is not a problem, but later it becomes a dept since the size of the parameters being managed becomes larger and larger.

Second, with an independent process, it is hard to conduct a series of experiments that utilizes persistent data among the experiments. For instance, let us consider an experiment to conduct a series of rendering jobs with several different parameters against the same scene. Naturally, we want to use the scene data as a persistent date among the executions to save scene loading time. Unfortunately, in Version 2, it is not possible because we always need to restart the renderer process on every executions.

Version 3 – Renderer as a Library. To resolve the aforementioned problem, we decided to redesign the renderer as *as a library* (Figure. 1.b). In Version 3, all operations to the internal state of the renderer, such as modifying the scene or organizing rendering jobs, must be conducted by a script with provided API, not by a scene definition file. As a matter of course, the scene definition file is deprecated in Version 3. This decision is based on the observation of the requirements in the actual research environment where the renderer is mostly used with experimental codes. This modification could remove the necessity of the intermediate IO layers to communicate with a renderer process, which eventually contributed to the simpleness of the experimental code. Note that this change might narrow down the possible volume of users, because we always need to write codes to render even the simplest scene. I believe, however, we could compensate this possibility because we could expect the users of the framework - developers and researchers - would anyways be familiar with coding.

2.2 Orchestration with Python

Python API. In Version 3, the features of the renderer can be directly accessible via Python API so that we can directly manipulate the renderer from the experimental codes. For instance you can manipulate the scene or organize the renderer jobs only with the Python API. Although we also expose C++ API as well as Python API, our

[4]`https://jinja.palletsprojects.com/en/2.10.x/`

```cpp
#include <lm/lm.h>

int main(int argc, char** argv) {
    // Initialize the framework
    lm::init();

    // Film for the rendered image
    lm::asset("film1", "film::bitmap", {
        {"w", 1920}, {"h", 1080}});
    // Pinhole camera
    lm::asset("camera1", "camera::pinhole", {
        {"position", {5.101118, 1.083746, -2.756308}},
        {"center", {4.167568, 1.078925, -2.397892}},
        {"up", {0,1,0}},
        {"vfov", 43.001194}});
    // OBJ model
    lm::asset("obj1", "model::wavefrontobj", {
        {"path", "<path␣to␣.obj␣file>"}});

    // Camera
    lm::primitive(lm::Mat4(1), {
        {"camera", lm::asset("camera1")}});
    // Create primitives from model asset
    lm::primitive(lm::Mat4(1), {
        {"model", lm::asset("obj1")}});

    // Build acceleration structure and render
    lm::build("accel::sahbvh");
    lm::render("renderer::pt", {
        {"output", lm::asset("film1")},
        {"spp", 10},
        {"maxLength", 20}});

    // Save rendered image
    lm::save(lm::asset("film1"), opt["out"]);

    return 0;
}
```

```python
import lightmetrica as lm


# Initialize the framework
lm.init()

# Film for the rendered image
lm.asset('film1', 'film::bitmap', {
    'w': 1920, 'h': 1080})
# Pinhole camera
lm.asset('camera1', 'camera::pinhole', {
    'position': [5.101118, 1.083746, -2.756308],
    'center': [4.167568, 1.078925, -2.397892],
    'up': [0,1,0],
    'vfov': 43.001194})
# OBJ model
lm.asset('obj1', 'model::wavefrontobj', {
    'path': '<path␣to␣.obj␣file>')})

# Camera
lm.primitive(lm.identity(), {
    'camera': lm.asset('camera1')})
# Create primitives from model asset
lm.primitive(lm.identity(), {
    'model': lm.asset('obj1')})

# Build acceleration structure and render
lm.build('accel::sahbvh', {})
lm.render('renderer::pt', {
    'output': lm.asset('film1'),
    'spp': 10,
    'maxLength': 20})

# Save rendered image
lm.save(lm.asset('film1')
```

Figure 2: Equivalent code to render an image using C++ API (left) and Python API (right). The API calls for each language are aligned in the same lines (e.g., lm::init() and lm.init()).

recommendation is to use Python API as much as possible. Figure 2 shows an example usage of the API in either of C++ or Python. I believe our Python API is flexible enough to cover the most of experimental workflow. Technically, we can even extend the renderer only with Python API, e.g., extending rendering techniques or materials (Figure. 3), although we do not recommend it due to the massive performance loss.

Also, our Python API supports automatic type conversion of the types between the corresponding Python and C++ interfaces. Some of the internal types appeared in the C++ API are automatically converted to the corresponding types in Python. For instance, math types (`vec3`, `mat4`) are converted to numpy array and `Json` type is converted to a `dict`.

Jupyter Notebook Extension. In order to boost the productivity of the experiments, we also provide a Jupyter notebook[5] extension so that we can implement various experiments interactively in the notebook. For instance, the examples and functinal tests of the framework are implemented within the Jupyter notebook[6]. The generated notebooks are also integrated into the online documentation.

```cpp
#include <lm/lm.h>

LM_NAMESPACE_BEGIN(LM_NAMESPACE)

class Renderer_AO final : public Renderer {
private:
  Film* film_;
  long long spp_;

public:
  virtual bool construct(const Json& prop) override {
    film_ = comp::get<Film>(prop["output"]);
    if (!film_) {
      return false;
    }
    spp_ = prop["spp"];
    return true;
  }

  virtual void render(const Scene* sc) const override {
    const auto [w, h] = film_->size();
    parallel::foreach(w*h, [&](long long idx, int tid) {
      thread_local Rng rng(42 + tid);
      const int x = int(idx % w);
      const int y = int(idx / w);
      const auto ray = sc->primaryRay(
        {(x+.5_f)/w, (y+.5_f)/h}, film_->aspectRatio());
      const auto hit = sc->intersect(ray);
      if (!hit) {
        return;
      }
      auto V = 0_f;
      for (long long i = 0; i < spp_; i++) {
        const auto [n, u, v] =
          hit->geom.orthonormalBasis(-ray.d);
        const auto d = math::sampleCosineWeighted(rng);
        V += sc->intersect(
          {hit->geom.p, u*d.x+v*d.y+n*d.z}, Eps, .2_f)
            ? 0_f : 1_f;
      }
      V /= spp_;
      film_->setPixel(x, y, Vec3(V));
    });
  }
};

LM_COMP_REG_IMPL(Renderer_AO, "renderer::ao");

LM_NAMESPACE_END(LM_NAMESPACE)
```

```python
import numpy as np
import lightmetrica as lm

@lm.pylm_component('renderer::ao')
class Renderer_AO(lm.Renderer):




  def construct(self, prop):
    self.film = \
      lm.Film.castFrom(lm.comp.get(prop['output']))
    if self.film is None:
      return False
    self.spp = prop['spp']
    return True


  def render(self, scene):
    w = self.film.size().w
    h = self.film.size().h
    rng = lm.Rng(42)
    lm.progress.start(w*h)
    def process(index, threadid):
      x = index % w
      y = int(index / w)
      rp = np.array([(x+.5)/w, (y+.5)/h])
      ray = scene.primaryRay(rp, self.film.aspectRatio())
      hit = scene.intersect(ray)
      if hit is None:
        return
      V = 0
      for i in range(self.spp):
        n, u, v = hit.geom.orthonormalBasis(-ray.d)
        d = lm.math.sampleCosineWeighted(rng)
        r = lm.Ray(hit.geom.p, np.dot(d, [u,v,n]))
        if scene.intersect(r, lm.Eps, .2) is None:
          V += 1
      V /= self.spp
      self.film.setPixel(x, y, np.full(3, V))
      lm.progress.update(y*w+x)
    lm.parallel.foreach(w*h, process)
    lm.progress.end()
```

Figure 3: Code examples to implement equivalent renderer extensions using C++ API (left) and Python API (right). Both codes implement simple ambient occlusion renderers. Although in practice C++ version is recommended in terms of the performance due to slow interpreter and single-threaded execution, Python version is better for prototyping because it doesn't require recompilation upon updates.

2.3 Improved Build Process

Refactoring CMake Scripts. The build system of Lighmetrica is organized by CMake scripts. In Version 3, we refactored the scripts with modern design patterns. The modifications include: target-based configuration, handling of the transitive dependencies (`PUBLIC`/`PRIVATE` targets), `INTERFACE` target for header-only dependencies, exported targets along with installation, and a preference to the Config mode package finding.

Improved Dependency Management. Especially, we improved the dependency management of the library. CMake can find external package with `find_package` command. This command finds the package based on the two different modes: Module mode and Config mode. To explain the difference, let's assume we have an application

[5] https://jupyter.org/

[6] https://github.com/lightmetrica/lightmetrica-v3/tree/master/functest

A and its dependent library B. In Module mode, the application A finds the library B using `Find*.cmake` file (find-module). CMake distribution contains `Find*.cmake` files for famous libraries (e.g., boost) yet in most cases the developer of the application needs to provide the file. On the other hand, in Config mode, the application A finds the library B based on `*Config.cmake` file distributed along with the library. To support Config mode, the library needs to generate and install `*Config.cmake` file in the specific relative location in the installation directory. In other words, the difference between two modes is that in Module mode the application is responsible for finding the library, on the other hand in Config mode, the library is responsible for being found by the application.

In Version 2, we used Module mode packages to find dependent libraries. We found, however, that using Module mode complicates the build process if the library is used as a library of other application because the application also needs to resolve the transitive dependencies of the library. For instance, this can happen in our case, e.g., when an experiment written in C++ (e.g., interactive visualization) wants to reference Lightmetrica as a dependent library. This means the library needs to distribute find-modules of the transitive dependencies as well as the module for the library itself. As a result, the developer of the application always needs to prepare for find-modules of the all dependencies including transitive dependencies.

In Version 3, on the contrary, we decided to use Config mode packages which can faithfully handle the transitive dependencies. Also, since Config mode packages are distributed along with the library distribution, the user does not need to prepare for the find-modules. To use Config mode packages throughout the build chain, we choose the dependencies which supports to export `*Config.cmake` files. If not supported, we patched the libraries.

Supporting add_submodule. On a small modification, Lightmetrica Version 3 can inject the library into the user application using *either* `find_package` command or `add_submodule` command. The latter approach is especially useful when you want to develop both experimental code and the framework managed in the different repository in the same time.

Installing Dependencies. In Version 2, all library dependencies must be installed by a developer using pre-built binaries or platform-dependent package manager or libraries built from the sources. As a result, the installation of the dependencies differs in accordance with the platform. For instance, we provided a pre-built binaries for Windows environment but in Linux environment the developer needs to build the libraries from the source.

In Version 3, on the other hand, the dependencies can be installed via *conda* package manager[7] irrespective to the platform. Conda is a package manager and environment management system being mainly used in the Python community. Although conda is mainly used to distribute Python packages, it is in fact a language-agnostic package manager. For instance, we can introduce C++ libraries such as boost with conda. Combined with config-mode packages of CMake, the installed libraries can

[7] `https://docs.conda.io/en/latest/`

be easily integrated into the application configured with CMake. In this version, we could successfully export the all the necessary library dependencies to conda. This means the build-time dependencies can only be installed with one conda command. The actual dependencies are listed in `environment.yml` file in the repository[8].

Also, conda can manage its own separated environments where you can manage different sets of the conda packages. It is useful when we want to combine different versions of the library in the same machine. For instance, two different versions of the framework might depend on the different versions of the library. In this case, it is not feasible to maintain both versions in the system-wide package manager, but separating the environment can manage two different versions simultaneously. It is especially important in research context since this situation often happens for instance when we need to maintain different experiments written for the different projects against the different versions of the framework.

Example: Building Framework From Source. As a result, the build process of the framework is drastically simplified. You can build the framework only with the following commands. For detail, please visit the documentation[9].

```
# ----- Clone the repository (including submodules)
$ git clone --recursive https://github.com/lightmetrica/lightmetrica-v3
$ cd lightmetrica-v3
# ----- Create a conda environment with dependencies
$ conda env create -f environment_{win,linux}.yml
$ conda activate lm3_dev
# ----- Build the framework
$ mkdir build && cd build
# ----- In Windows
$ cmake -G "Visual␣Studio␣15␣2017␣Win64" ..
$ start lightmetrica.sln
# ----- In Linux
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make -j
```

2.4 Extended Tests

Tests in Lightmetrica. Lightmetrica implements various tests to maintain the quality of the framework. In Version 3, we are using (or planning to use) four types of the tests: unit test, functional test, performance test, and statistical tests, whereas Version 2 only provided unit tests. All tests are designed to be automated and integrated into continuous integration (CI) services. Using CI services, we can check the consistency of the tests for every commit to the repository. If a commit breaks a test, the developer is immediately notified the failure. Also, some of the tests that is not determined by assertions are written in Jupyter notebooks and directly integrated into the documentation after the execution in the CI service.

[8] https://github.com/lightmetrica/lightmetrica-v3/blob/master/environment_win.yml

[9] https://lightmetrica.github.io/lightmetrica-v3-doc/build.html

Unit test. Unit testing is responsible for the tests of individual components of the framework. We mainly use unit testing to test internal features like component object manipulation, assets management, or scene management. In addition to the tests of the internal features written in C++, we also provide unit tests for the Python binding of the framework.

Functional test. Functional testing is responsible for testing behavioral aspects of the framework without requiring internal knowledge of the framework. In other words, functional testing is in charge for the tests of possible usage by users using only the exposed APIs. In software development, functional testing is coined as a test to check if software or library functions properly according to the requirements. This means we can use functional tests to define the requirements of the framework.

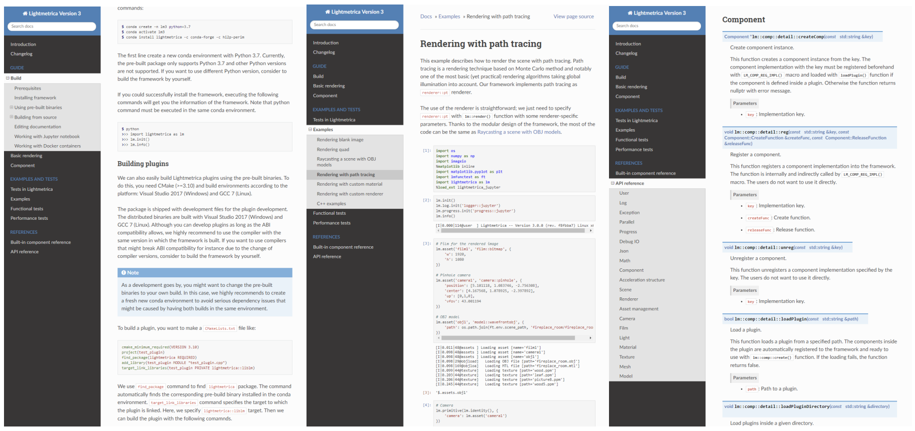
Performance test. Performance testing is responsible for testing performance requirements of the framework. We rather focus on testing relative performance rather than absolute performance, because relative performance comparison is what research and development requires. We mainly use the tests to compare performances of multiple implementations of the same interface, for instance, the performance of acceleration structures, scene loading, etc.

Statistical test. Statistical testing is responsible for testing correctness of the behavior of the framework using statistical approaches. We can describe statistical test being a kind of functional test where the correctness of the behavior is evaluated only by the statistical verification. We used the tests to verify the correctness of sampling functions or rendering techniques.

2.5 Documentation

Documentation of Lightmetrica. Documentation is crucial for usability of the framework. In Version 2, we combined a static site generator (Hugo[10]) and a generated documentation from the C++ source using Doxygen[11]. In Version 3, since we redeveloped the framework as a library mainly used by Python interface, we decided to use *Sphinx*[12] which is famous in Python community. Although Sphinx mainly targets the documenetation of Python libraries, it can be used to generate any forms of documentation. Sphinx adopts reStructuredText[13] as a markup language where we can write Markdown documents as simple and straightforward as e.g. Markdown. Sphinx can even extend its features with extensions. For instance, using *breath*[14] extension can allow us to integrate Doxygen-generated documentation inside the Sphinx documentation, or with *sbsphinx*[15] extension we can import Jupyter notebook as a sphinx

[10]https://gohugo.io/
[11]http://www.doxygen.nl/
[12]http://www.sphinx-doc.org/en/master/
[13]http://docutils.sourceforge.net/rst.html
[14]https://breathe.readthedocs.io/en/latest/
[15]https://nbsphinx.readthedocs.io/en/latest/



commands:

```
$ conda create -n lm3 python=3.7
$ conda activate lm3
$ conda install lightmetrica -c conda-forge -c hi2p-perim
```

The first line create a new conda environment with Python 3.7. Currently, the pre-built package only supports Python 3.7 and other Python versions are not supported. If you want to use different Python version, consider to build the framework by yourself.

If you could successfully install the framework, executing the following commands will get you the information of the framework. Note that python command must be executed in the same conda environment.

```
$ python
>>> import lightmetrica as lm
>>> lm.init()
>>> lm.info()
```

Building plugins

We can also easily build Lightmetrica plugins using the pre-built binaries. To do this, you need CMake (>=3.10) and build environments according to the platform: Visual Studio 2017 (Windows) and GCC 7 (Linux).

The package is shipped with development files for the plugin development. The distributed binaries are built with Visual Studio 2017 (Windows) and GCC 7 (Linux). Although you can develop plugins as long as the ABI compatibility allows, we highly recommend to use the compiler with the same version in which the framework is built. If you want to use compilers that might break ABI compatibility for instance due to the change of compiler versions, consider to build the framework by yourself.

Note

As a development goes by, you might want to change the pre-built binaries to your own build. In this case, we highly recommends to create a fresh new conda environment to avoid serious dependency issues that might be caused by having both builds in the same environment.

To build a plugin, you want to make a `CMakeLists.txt` file like:

```
cmake_minimum_required(VERSION 3.10)
project(test_plugin)
find_package(lightmetrica REQUIRED)
add_library(test_plugin MODULE "test_plugin.cpp")
target_link_libraries(test_plugin PRIVATE lightmetrica::liblm)
```

We use `find_package` command to find `lightmetrica` package. The command automatically finds the corresponding pre-build binary installed in the conda environment. `target_link_libraries` command specifies the target to which the plugin is linked. Here, we specify `lightmetrica::liblm` target. Then we can build the plugin with the following comamnds.

Rendering with path tracing

This example describes how to render the scene with path tracing. Path tracing is a rendering technique based on Monte Carlo method and notably one of the most basic (yet practical) rendering algorithms taking global illumination into account. Our framework implements path tracing as `renderer::pt` renderer.

The use of the renderer is straightforward; we just need to specify `renderer::pt` with `lm::render()` function with some renderer-specific parameters. Thanks to the modular design of the framework, the most of the code can be the same as Raycasting a scene with OBJ models.

```
[1]: import os
     import numpy as np
     import imageio
     %matplotlib inline
     import matplotlib.pyplot as plt
     import lmfunctest as ft
     import lightmetrica as lm
     %load_ext lightmetrica_jupyter

[2]: lm.init()
     lm.log.init('logger::jupyter')
     lm.progress.init('progress::jupyter')
     lm.info()

     [I|0.000|114@user  ] Lightmetrica -- Version 3.0.0 (rev. f8fbba7) Linux x

[3]: # Film for the rendered image
     lm.asset('film1', 'film::bitmap', {
         'w': 1920,
         'h': 1080
     })

     # Pinhole camera
     lm.asset('camera1', 'camera::pinhole', {
         'position': [5.101118, 1.083746, -2.756308],
         'center': [4.167568, 1.078925, -2.397892],
         'up': [0,1,0],
         'vfov': 43.001194
     })

     # OBJ model
     lm.asset('obj1', 'model::wavefrontobj', {
         'path': os.path.join(ft.env.scene_path, 'fireplace_room/fireplace_roo
     })

     [I|0.011|48@assets ] Loading asset [name='film1']
     [I|0.098|48@assets ] Loading asset [name='camera1']
     [I|0.098|48@assets ] Loading asset [name='obj1']
     [I|0.098|29@objload]   Loading OBJ file [path='fireplace_room.obj']
     [I|0.098|169@objloa]   Loading MTL file [path='fireplace_room.mtl']
     [I|0.099|44@texture]   Loading texture [path='wood.ppm']
     [I|0.203|44@texture]   Loading texture [path='leaf.ppm']
     [I|0.206|44@texture]   Loading texture [path='picture8.ppm']
     [I|0.245|44@texture]   Loading texture [path='wood5.ppm']

[3]: '$.assets.obj1'

[4]: # Camera
     lm.primitive(lm.identity(), {
         'camera': lm.asset('camera1')
     })
```

Component

`Component *lm::comp::detail::createComp(const std::string &key)`

Create component instance.

This function creates a component instance from the key. The component implementation with the key must be registered beforehand with `LM_COMP_REG_IMPL()` macro and loaded with `loadPlugin()` function if the component is defined inside a plugin. Otherwise the function returns nullptr with error message.

Parameters

- `key` : Implementation key.

`void lm::comp::detail::reg(const std::string &key, const Component::CreateFunction &createFunc, const Component::ReleaseFunction &releaseFunc)`

Register a component.

This function registers a component implementation into the framework. The function is internally and indirectly called by `LM_COMP_REG_IMPL()` macro. The users do not want to use it directly.

Parameters

- `key` : Implementation key.
- `createFunc` : Create function.
- `releaseFunc` : Release function.

`void lm::comp::detail::unreg(const std::string &key)`

Unregister a component.

This function unregisters a component implementation specified by the key. The users do not want to use it directly.

Parameters

- `key` : Implementation key.

`bool lm::comp::detail::loadPlugin(const std::string &path)`

Load a plugin.

This function loads a plugin from a specified path. The components inside the plugin are automatically registered to the framework and ready to use with `lm::comp::create()` function. If the loading fails, the function returns false.

Parameters

- `path` : Path to a plugin.

`void lm::comp::detail::loadPluginDirectory(const std::string &directory)`

Load plugins inside a given directory.

Figure 4: Documentation of Lightmetrica Version 3. The documentation is generated by Sphinx documentation generator. The guide (left) explains the basic to advanced usage of the framework. The examples and tests (middle) shows the actual usage of the framework with Python or C++ API. These pages are generated directly from the executed Jupyter notebook. The references (right) shows comprehensible references of API and build-in components. These pages are generated from the source code comments.

documentation. We automated the generation of the documentation and the deployment to GitHub sites using a CI service.

Types of Documentation. We have several types of documentations: *guides*, *examples and tests*, and *references*. The *guide* explains basic to advanced usage of the framework, for instance how to build the framework from the source or how to render the images, or how to extend the framework.

The *example and tests* illustrates actual usage of the framework using Python/C++ API and the results of functional/performance/statistics tests. The examples and tests are written in a Jupyter notebook and the results are always executed in a CI build. It guarantees that the results are always generated from the working version of the framework. In other words, the broken documentation indicates a new commit ruined the runtime behavior of the framework.

The *references* describe the explanation of the detailed API and the components of the framework. *API reference* describes function-wise or interface-wise features of the framework. It is especially useful when you want develop your own extensions. *Build-in component references* describes the built-in components of the interfaces such as materials or renderers. It also explains a brief background of the theory behind the component. This reference is useful when you want to use the renderer as a user.

2.6 Improved Component Object System

Component Object System in Lightmetrica. Lightmetrica is build upon a component object system which provides various features to support extensibility as well as usability based on object-oriented paradigm. All extensible features of the framework, e.g., materials or renderers etc., are implemented based on this system. The purpose of our component object system is to provide a complete decoupling between the interface and the implementation. Our component interface is based on the type erasure by virtual classes in C++. Once an instance is created, the type of the implementation is erased and we can access the underlying implementation through the interface type.

Creating Instances with Factory. To create an instance in C++, typically, we need to know the type of the derived class. For instance, assume we have an interface A (pure virtual class) and an implementation A1 inheriting A. Here, to instantiate A1, we need an definition of the A1, e.g., by including the header containing the definition of A1. A problem is that we pointlessly increase the coupling between A1 and the creator of the class, although once created they are only used through the interface A. This requires to expose the header containing A1 as a separated file, where the header needs to contain private members being never referenced in the installation of the class unless we use the trick like pimpl.

To resolve this problem, we adopted a common practice to use an abstract factory. Instead of using actual type of the derived class, an instance is created by an (string) identifier. The factory need to know the mapping between the identifier and the way of creating instance of an implementation. Our component object system can automate the registration process by a simple macro. Furthermore, our component system is flexible enough to implement plugins as easily as built-in components.

Differences Between Version 2 and 3. In Version 3, we refactored the already-implemented features in the component object system in Version 2. Furthermore, some features are newly implemented but some other features are deprecated due to a design choice.

Particularly in Version 3, we deprecated portable interface support. This feature allows the user to extend the interface irrespective to ABI of the compilers and standard libraries. To achieve this, we needed to reimplement our own virtual function mechanism where a function call is automatically converted to a function call with portable c-interfaces in compile time. Although the feature worked great as expected, we decided to deprecate the feature with the following reasons. First, to describe the component interface and implementation, a developer needed to write boilerplate codes using C++ macros, which has lessened the maintainability of the codes. Second, this feature could be an cause of massive performance loss because the reimplemented virtual function mechanism could prevent the optimization by compilers (e.g., devirtualization). Last but not least, this feature was rarely used in the actual research projects because in most cases the developer wants to build the framework from the source and doesn't care about the binary portability issues. In Version 3, the interface uses standard virtual function mechanism in C++.

Newly implemented features include the handling of component hierarchy, query mechanism, and serialization support. In the rest of this section, we will explain the details of the features.

Creating Interface. A *component interface* is simply an C++ virtual class that directly/indirectly inherits `lm::Component` class. Note that `lm::Component` can also be a component interface. An example of the component interface with a virtual function would be like:

```
// Version 3 component interface
struct TestComponent : public lm::Component {
   virtual int f() = 0;
};
```

Implementing Interface. *Implementing* a component interface is same as C++ standard in a sense that the user needs to inherit an interface type and override the virtual functions. The implemented class must be registered to the framework via `LM_COMP_REG_IMPL()` macro. For instance, implementing TestComponent above looks like

```
struct TestComponent_Impl final : public TestComponent {
    virtual int f() override { ... }
}

LM_COMP_REG_IMPL(TestComponent_Impl, "testcomponent::impl");
```

where `LM_COMP_REG_IMPL()` macro takes a class name in the first argument and the identifier of the class in the second argument. The identifier is just a string and we can specify any letters in it, yet as a convention, we used `interface::impl` format throughout the framework. Note that the registration process happens in static initialization phase. If an identifier is already registered, the framework reports a runtime error.

Creating Plugin. Creating plugins of the framework is straightforward. A plugin is just an component implementation placed in the dynamically loadable context. This is flexible because the user do not need to care about the change of the syntax to create an plugin. An dynamic libraries can contain as many component implementation as you want. A plugin can be loaded by `lm::comp::loadPlugin()` function and unloaded by `lm::comp::unloadPlugins()` function.

Creating Instance. Once a registration has done, we are ready to use it. We can create an instance of a component by `lm::comp::create()` function. For instance, creating `testcomponent::impl` component reads

```
const auto comp =
    lm::comp::create<TestComponent>("testcomponent::impl", "");
```

The first argument is the identifier of the implementation, the second argument is the component locator of the instance if the object is integrated into the global component hierarchy. For now, let's keep it empty. You need to specify the type of the component interface with template type. If the instance creation fails, the function will return nullptr. `lm::comp::create()` function returns `unique_ptr` of the specified interface type. The lifetime management of the instance is up to users. The `unique_ptr` is equipped with a custom deleter to support the case where the instance is created in the different dynamic libraries.

Parameterized Creation. We can pass arbitrary arguments in a JSON-like format as a third argument of lm::comp::create() function. We depend nlohmann/json[16] library to achieve this feature. See the link for the supported syntax and types.

```
const auto testcomp =
    lm::comp::create<TestComponent>(
        "testcomponent::impl_with_param", "", {
            {"param1", 42},
            {"param2", "hello"}
        });
```

The parameters are routed to `lm::Component::construct()` function implemented in the specified component. We can extract the values from the Json type using successors like STL containers. For convenience, we provided serializers to automatically convert types to/from the JSON type, which includes e.g. vector / matrix types, raw pointer types.

```
struct TestComponent_ImplWithParam final : public TestComponent {
    virtual bool construct(const lm::Json& prop) override {
        const int param1 = prop["param1"];
        const std::string param2 = prop["param2"];
        ...
        return true;
    }
    virtual int f() override { ... }
}

LM_COMP_REG_IMPL(
    TestComponent_ImplWithParam, "testcomponent::impl_with_param");
```

Component Hierarchy And Locator. Composition of the `unique_ptr` of components or raw pointers inside a component implicitly defines a component hierarchy of the components. In the framework, we adopts a strict ownership constraint that one instance of the component can only be possessed and managed by a single another component. In other words, we do not allow to use `shared_ptr` to manage the instance of the framework. This constraint makes it possible to identify a component inside the hierarchy by a locator.

A component locator is a string to uniquely identify an component instance inside the hierarchy. The string start with the character `$` and arbitrary sequence of characters separated by '.' (dot character). For instance, `$.assets.obj1.mesh1`. Each string separated by '.' is used to identify the components owned by the current node inside the hierarchy. By iteratively tracing down the hierarchy from the root, the locator can identify an single component instance.

When we create an instance, we can also specify the component locator in the second argument. An helper function `lm::Component::makeLoc()` is useful to make locator appending to the current locator. For instance, the following creation of an instance called inside `lm::Component::construct()` function of a component with locator `$.test` will create a component with locator `$.test.test2`.

[16]https://github.com/nlohmann/json

```
struct TestComponent_Container final : public lm::Component {
    Ptr<lm::Component> comp;
    virtual bool construct(const lm::Json& prop) override {
        // Called inside a component with locator = $.test,
        // create an instance with locator = $.test.comp
        comp = lm::comp::create<lm::Component>(
             "testcomponent::nested", makeLoc("comp"));
        return true;
    }
    virtual Component* underlying(const std::string& name) const override {
        // Underlying component must be accessible
        // with the same name specified in create function
        return name == comp->name() ? comp.get() : nullptr;
    }
};
```

Also, the underlying component must be accessible by the specified name using `lm::Component::underlying()` function. `lm::Component::name()` function is useful to extract the name of the component. Once the above setup completes, we can access the underlying component globally by `lm::comp::get()` function.

```
const auto comp = lm::comp::get<lm::Component>("$.test.comp");
```

Note that some advanced features like serialization are based on this mechanism. Even if it seems to be working without ill-formed components, e.g., those not specifying locator or not implementing `lm::Component::underlying()` function, it will definitely break some feature in the end.

Weak References. A raw pointer composed inside a component is handled as a weak reference to the other (owned) components. Our framework only allows weak reference as a back edge (the edge making cycles) in the component hierarchy. A weak reference is often used as an injected instance to the other components within `lm::Component::construct()` function.

For instance, the following component accepts `ref` parameter as a string representing the locator of the component. We can then inject the weak reference using `lm::comp::get()` function.

```
struct TestComponent_WeakRef1 final : public lm::Component {
    lm::Component* ref;
    virtual bool construct(const lm::Json& prop) override {
        ref = lm::comp::get<lm::Component>(prop["ref"]);
        return true;
    }
};
```

Alternatively, one can inject the raw pointer directly to the component. because the pointer types are automatically serialized to JSON type. This strategy is especially useful when we want to inject the pointer of the type inaccessible from the component hierarchy.

```
const lm::Component* ref = ...
const auto comp = lm::comp::create<lm::Component>(
    "testcomponent::weakref2", "", {
        {"ref", ref}
    });
```

where

```
struct TestComponent_WeakRef2 final : public lm::Component {
    lm::Component* ref;
    virtual bool construct(const lm::Json& prop) override {
        ref = prop["ref"];
        return true;
    }
};

LM_COMP_REG_IMPL(
    TestComponent_WeakRef2, "testcomponent::weakref2");
```

Querying Information. A component provides a way to query underlying components. The framework utilizes this mechanism to implement some advanced features. To support querying of the underlying components, a component must implement both `lm::Component::underlying()` and `lm::Component::foreachUnderlying()` functions.

`lm::Component::underlying()` function return the component with a query by name. `lm::Component::foreachUnderlying()` function on the other hands enumerates all the underlying components. visit function needs to distinguish both `unique_ptr` (owned pointer) and raw pointer (weak reference) in the second argument. Yet `lm::comp::visit()` function will call them automatically according to the types for you. For instance, for the component containing `unique_ptr`:

```
struct TestComponent_Container1 final : public lm::Component {
    std::vector<Ptr<lm::Component>> comps;
    std::unordered_map<std::string, int> compMap;
    virtual Component* underlying(const std::string& name) const override {
        return comp.at(compMap.at(name)).get();
    }
    virtual void foreachUnderlying(const ComponentVisitor& visit) override {
        for (auto& comp : comps) {
            lm::comp::visit(visit, comp);
        }
    }
};
```

Similary, for the component containing weak references:

```
struct TestComponent_Container2 final : public lm::Component {
    lm::Component* ref1;
    lm::Component* ref2;
    virtual Component* underlying(const std::string& name) const override {
        if (name == "ref1") { return ref1; }
        if (name == "ref2") { return ref2; }
        return nullptr;
    }
    virtual void foreachUnderlying(const ComponentVisitor& visit) override {
        lm::comp::visit(visit, ref1);
        lm::comp::visit(visit, ref2);
    }
};
```

Supporting Serialization. Our serialization feature depends on *cereal* library[17]. Yet unfortunately, a polymorphism support of cereal library is restricted because the declaration of the derived class must be exposed to the global. In our component object system, an implementation is completely separated from the interface and there is no way to find corresponding implementation automatically.

We workaround this issue to implement serialization for a specific archive and route the object finding mechanism of cereal to use these functions. To enable the feature, a component must implement `lm::Component::save()` and `lm::Component::load()` functions. This means we can no longer use arbitrary archive type, but I believe it is a reasonable compromise. The default archive type is defined as `lm::InputArchive` and `lm::OutputArchive`.

Implementing almost-similar two virtual functions is cumbersome. To mitigate this, we provided `LM_SERIALIZE_IMPL()` helper macro. The following code serializes member variables including component instances, or weak references. Note that we can even serialize raw pointers, as long as they are weak references pointing to a component inside the component tree, and accessible by component locator.

```
struct TestComponent_Serial final : public lm::Component {
    int v;
    std::vector<Ptr<lm::Component>> comp;
    lm::Component* ref;
    LM_SERIALIZE_IMPL(ar) {
        ar(v, comp, ref);
    }
};
```

Singleton. A component can be used as a singleton, and our framework implemented globally-accessible yet extensible features using component as singleton. For convenience, we provide `lm::comp::detail::ContextInstance` class to make any component interface a singleton.

3 Conclusion

In this article, we described various design choices of Lightmetrica Version 3, a research-orieitend renderer that I have developed since 2014. In Version 3, we redesigned the framework as a practical environment that can be used for the research and development of renderers, by redesigning and rewriting the most of features from the ground up. In this article, we have rather focused on the basic features to support the implementation of the renderer and experiments. In the next article, if exists, I want to describe the improvements of the rendering features thereof. Lightmetrica Version 3 is still under development and your contribution is always welcome. I hope this article could inspire you.

[17]`https://uscilab.github.io/cereal/`