About

The goal of this un-graded assignment is to give a quick tutorial on how to download, compile, and execute NORI the rendering framework that will be used during the course. Step-by-step instructions will be given for setting up the project, and on implementing a first `integrator` to generate a first image with the software.

NORI is a minimalistic ray tracer written in C++ by Prof. Wenzel Jakob (EPFL, Switzerland). It runs on Windows, Linux, and Mac OS and provides a foundation for the programming assignments in the Modeling and Simulation of Appearance course. While NORI provides much support code to simplify your development work as much as possible, the code that you will initially receive from us does very little: It loads a scene and saves a rendered image as both a PNG and OpenEXR image, but the actual rendering code is missing, hence the output image just consists of black pixels. Your task will be to extend this system to a full-fledged physically-based renderer as part of programming assignments and your final project.

1 Nori core features

The NORI base code provides many features that would be tedious to implement from scratch. The following are included:

- A simple GUI to watch images as they render
- An XML-based scene file loader
- Basic point/vector/normal/ray/bounding box classes
- A pseudorandom number generator (PCG32)
- Support for saving output as OpenEXR files
- A loader for Wavefront OBJ files
- Support for textures in all materials
- An optimized bounding volume hierarchy builder
- Ray-triangle intersection

- Code for multi-threaded rendering
- Image reconstruction filters

2 References

You may find the following general references useful:

- *"Physically Based Rendering, Third Edition: From Theory To Implementation"*[1] by Matt Pharr, Wenzel Jakob, and Greg Humphreys. Morgan Kaufmann, 3rd edition, Nov 2016. http://www.pbr-book.org/
- *"Robust Monte Carlo Methods for Light Transport Simulation"*[2], PhD Thesis by Eric Veach, Stanford University, December 1997. https://graphics.stanford.edu/papers/veach_thesis/thesis-bw.pdf
- *"Advanced Global Illumination"* by Philip Dutre, Philippe Bekaert, and Kavita Bala. AK Peters, 2nd edition, August 2006.
- *"Global Illumination Compendium - The Concise Guide to Global Illumination Algorithms"*, Philip Dutre, 2003. https://people.cs.kuleuven.be/~philip.dutre/GI/

Feel free to consult additional references when completing projects, but remember to cite them in your writeup. When asked to implement feature X, we request that you don't go and read the source code of the implementation of X in some other renderer, because you will likely not learn much in the process. The PBRT book is excluded from this rule. If in doubt, get in touch with the course staff.

3 Instructions & Guidelines

There will be a number of programming assignments throughout the quarter where you will build up specific functionality in your render, followed by a final (programming) project at the end of the quarter. The assignments have to be solved and submitted in groups of **max 2 people**. More information about the final project and the rendering competition will come later in the quarter. The deadlines are specified for each assignment. These are the dates to submit your exercises (code and report) through **Moodle**.

The goal of these exercises is to help you create the foundation of a good renderer for subsequent assignments and the final projects. It is your job to convince us that you have implemented the assignments correctly, through the results you will be submitting, and the

[1] Scientific and Technical Academy Award
[2] Scientific and Technical Academy Award

quality and readability of your code. We will deduct points for artifacts, low-quality renders, and lack of comparisons. Make sure to start working on the homework as early as possible. Building your own advanced renderer is a lot of work, but also a lot of fun.

In the following, we will give you some detailed instructions on how to setup NORI (Section 4), we will guide you through the code and the scene definition in NORI (Section 5), and we will end up creating very first simple renderer in NORI (Section 6).

4 Downloading and setting up Nori

The base code of NORI that we will use during the course is provided as a `zip` file through Moodle. In the following, we describe how to setting up the build system for each target platform using the CMake[3] build system. Tip: it's a good idea to set the build mode to Release unless you are tracking down a particular bug. The debug version runs much slower (by a factor of 50 or more).

> **CMake Troubleshooting:** After any error with CMake (e.g., because you were missing a library), remember to **remove CMakeCache.txt and CMakeFiles** in the build directory. Otherwise, CMake will keep failing even if you fix the problem.

Linux & Mac OS X Begin by installing CMake on your system. On Mac OS X, you will also need to install a reasonably up-to-date version of XCode along with the command line tools. On Linux, any reasonably recent version of GCC or Clang will work (except g++13). Navigate to the NORI folder, create a build directory and generate the CMake files. You can use either the GUI or the command line:

1. **Using CMake-GUI:**

```
$ cd path-to-nori
$ mkdir build
$ cd build
$ cmake-gui ..
```

2. **Using the command line:**

```
$ sudo apt install g++-12 cmake
$ cd path-to-nori
$ mkdir build
$ cd build
$ cmake -DCMAKE_CXX-COMPILER=/usr/bin/g++-12 ..
```

[3] https://cmake.org/download/

After the Makefiles are generated, simply run `make` to compile all dependencies and NORI itself.

```
$ make -j 4
```

This can take quite a while; the above command compiles with four processors at the same time. Note that you will probably see many warning messages while the dependencies are compiled–you can ignore them.

Windows / Visual Studio Begin by installing Visual Studio (VS) 2017 or newer (older versions won't do) and a reasonably recent ($\leq$3.x) version of CMake. Community versions of VS should work just fine.

You can use either CMake-GUI or the command line:

1. Using CMake-GUI:

Start CMake and navigate to the location where you extracted NORI. It is a good idea to choose a `build` directory that is different from the source directory. After setting up the project, click the *Configure* and *Generate* button. This will create a file called `nori.sln` – double-click it to open Visual Studio. The *Build→Build Solution* menu item will automatically compile all dependency libraries and Nori itself; the resulting executable is written to the Release or Debug subfolder of your chosen build directory. Note that you will probably see many warning messages while the dependencies are compiled – you can ignore them.

2. Using the command line:

Open a Powershell and navigate to the location where you extracted NORI. Generate the build files:

```
$ cd path-to-nori
$ cmake -G "Visual Studio 17 2022" -A x64 -B build
```

Then, each time you want to recompile run the following command (equivalent to make for Linux):

```
$ cmake --build build --config Release -j 4
```

5 Nori's high-level overview

The NORI repository consists of the base code files and several dependency external libraries. Both are explained in Table 1. All your work will be implemented in the folders `src` and `include/nori`. Take a moment to browse through the header files in `include/nori`. You will generally find all important interfaces and their documentation in this place. Most headers files also have a corresponding `.cpp` implementation file in the `src` directory.

The most basic class in NORI is called `NoriObject` (`include/nori/object.h`) – it is the base class of everything that can be constructed using the XML scene description language (see Section 5.1). All other objects in NORI (e.g. cameras, geometry, materials, integrators...) derive from this class, and expose additional more specific functionaliy. In addition, there are a few additional objects and headers that implement additional utility and mathematical functions, that will be used in the assignments.

All parameters of the scene being rendered, including emitters (light sources), geometry, and materials, are stored in the class `Scene`. It provides access to all elements in the scene, sampling rountiness (that you will need to implement in the first assignment), and visibility computations by mean of *ray tracing*. Take a careful look on the interface of this class on `include/nori/scene.h`.

Basic classes and structs NORI builds upon several classes that implement the basic elements of a renderer, in particular geometric data (i.e. points, vectors) and (spectral) radiance, as well as routines for sampling that you will be using during the next practical assignments. In particular, it is sensible to familiarize with the following classes:

- **`Vector3f`, `Point3f`, `Normal3f` (`include/nori/vector.h`):** The basic geometric structures encode vectors, points and normals for N-dimensional spaces, implemented as templatized vectors from the lineal algebra library `Eigen`. The actual structures we will be using are declared in `include/nori/common.h`. You can perform the basic scalar arithmetic operators over vectors, as well as accessing its elements using the operator `Vector3f::operator []`. For more details, check the documentation of Eigen[4].
- **`Frame` (`include/nori/frame.h`):** Stores a three-dimensional orthonormal coordinate frame; This class is mostly used to quickly convert between different Cartesian coordinate systems and to efficiently compute certain quantities. Note that all shading operations in NORI are performed on a local orthonormal frame centered on the surface's normal.
- **`Ray3f`(`include/nori/ray.h`):** Simple ray-segment data structure used for ray tracing, as well as for querying radiance from the scene. This structure includes the origin and direction of the ray (members `Ray::o` and `Ray::d` respectively.
- **`Color3f`(`include/nori/color.h`):** Simple structure to store real-valued RGB radiance. Similar to e.g. `Vector3f` it derives from `Eigen`'s arrays, and allows element-wise operations in vectorial form.
- **`Mesh` (`include/nori/mesh.h`):** As most renderers out there, NORI supports triangle meshes for representing geometry. Essentially, each object is discretized as a bunch of triangles, which you can access to from the class `Mesh`. Additionally, it links the geometry with its particular `Material` and, in case of emitting sur-

[4] https://eigen.tuxfamily.org/dox-3.2/

faces, its `Emitter`. You can access the material and emitters by using the functions `Mesh::getBSDF()` and `Mesh::getEmitter()`, respectively. For more details, please see the interface source.

- **`Intersection`(`include/nori/mesh.h`):** Structure that stores the local information of an intersection between a `Ray` and 3D geometry in the `Scene` (modeled as triangle meshes using the class `Mesh`, see below). It includes all information required for shading the intersection point, including the `Point3f` of intersection, the local `Frame`, the texture coordinates, and a pointer to the associated `Mesh` from which the material can be accessed. To obtain the intersection between a `Ray` and the geometry in the scene, you should use the function `Scene::rayIntersect()` from the `Scene` (see details in `include/nori/scene.h`).

Exposed interfaces The base interfaces that derive from `NoriObject`, and that will describe both the implemented rendering algorithms and scene properties are:

- **Integrator (`include/nori/integrator.h`):** The different rendering techniques are collectively referred to as integrators, since they perform integration over a high-dimensional space. Each integrator represents a specific approach for solving the light transport equation—usually favored in certain scenarios, but at the same time affected by its own set of intrinsic limitations. During the next assignments, you will be asked to implement a few of these integrators. Integrators have one main function, `Integrator:Li()`, that for a given `Scene` and `Ray` generated from the camera, it returns a sample of the radiance carried out by the ray, enconded in the class `Spectrum`.

- **Camera (`include/nori/camera.h`):** This class provides an abstract interface to cameras in Nori and exposes the ability to sample their response function by generating a ray to be traced towards the scene. By default, only the basic perspective (`src/perspective.cpp`) camera is implemented, but you can implement additional cameras in the final project.

- **Emitter (`include/nori/emitter.h`):** Superclass of all emitters (light sources) implemented in NORI. As all sampleable objects, it includes three main routines: a) `Emitter:sample()`, that samples a direction towards the emitter and stores it in a helper struct called `EmitterQueryRecord`; b) `Emitter:pdf()`, that returns the probability of generating a sample stored on an `EmitterQueryRecord`; and c) `Emitter:eval()`, that evaluates the light source for a sample stored in `EmitterQueryRecord`. More details on how to use `Emitter:sample()` will be found later in this assignment; while the rest will be used thoroughly in Assignment 2.

- **BSDF (`include/nori/bsdf.h`):** Superclass of all BSDF (appearance models) implemented in NORI. In the same way as `Emitter`, it is also a sampleable object and therefore implements three main routines: a) `BSDF:sample()`, that samples a di-

Directory	Description
src	A directory containing the main C++ source code
include/nori	A directory containing header files with declarations
ext	External dependency libraries (see below)
scenes	Example scenes and test datasets to validate your implementation
CMakeLists.txt	A CMake build file which specifies how to compile and link Nori
External dependencies	
Directory	**Description**
ext/openexr	A high dynamic range image format library
ext/pcg32	A tiny self-contained pseudorandom number generator
ext/filesystem	A tiny self-contained library for manipulating paths on various platforms
ext/pugixml	A light-weight XML parsing library
ext/tbb	Intel's Boost Thread Building Blocks for multi-threading
ext/tinyformat	Type-safe C++11 version of printf and sprintf
ext/hypothesis	Functions for statistical hypothesis tests (won't be used in our assignments)
ext/nanogui	A minimalistic GUI library for OpenGL
ext/nanogui/ext/eigen	A linear algebra library used by `nanogui` and NORI.
ext/zlib	A compression library used by OpenEXR

Table 1: *Directory high-level description, and* NORI *dependencies.*

rection and stores it in a helper struct called `BSDFQueryRecord`; b) `BSDF:pdf()`, that returns the probability of generating a sample stored on an `BSDFQueryRecord`; and c) `BSDF:eval()`, that evaluates the BSDF for a pair of directions stored in `BSDFQueryRecord`. More details on how to use `BSDF:eval()` will be found later in this assignment; the rest will be implemented and used in Assignment 2.

In Section 6 we show, step-by-step, how to implement your first physically-based `Integrator` for direct light (`DirectEmitterIntegrator`), as well as the most basic emitters (`PointLight`) and materials (`Diffuse`).

5.1 Scene file format and parsing

NORI uses a very simple XML-based scene description language, which can be interpreted as a kind of building plan: the parser creates the scene step by step as it reads the scene file from top to bottom. The XML tags in this document are interpreted as requests to construct certain C++ objects including information on how to put them together.

Each XML tag is either an object or a property. Objects correspond to C++ instances that will be allocated on the heap. Properties are small bits of information that are passed to an object at the time of its instantiation. For instance, the following snippet creates red diffuse BSDF:

```
<bsdf type="diffuse">
    <color name="albedo" value="0.5, 0, 0"/>
</bsdf>
```

Here, the <bsdf> tag will cause the creation of an object of type BSDF, and the type attribute specifies what specific subclass of BSDF should be used. The <color> tag creates a property of name albedo that will be passed to its constructor. If you open up the C++ source file src/diffuse.cpp, you will see that there is a constructor, which looks for this specific property:

```
Diffuse(const PropertyList &propList) {
    m_albedo = propList.getColor("albedo", Color3f(0.5f));
}
```

The piece of code that associates the "diffuse" XML identifier with the Diffuse class in the C++ code is a macro found at the bottom of the file:

```
NORI_REGISTER_CLASS(Diffuse, "diffuse");
```

You will use this macro for all the new code you deploy using NORI (see Section 6).

Certain objects can be nested hierarchically. For example, the following XML snippet creates a mesh that loads its contents from an external OBJ file and assigns a red diffuse BRDF to it.

```
<mesh type="obj">
    <string type="filename" value="bunny.obj"/>

    <bsdf type="diffuse">
        <color name="albedo" value="0.5, 0, 0"/>
    </bsdf>
</mesh>
```

Implementation-wise, this kind of nesting will cause a method named `addChild()` to be invoked within the parent object. In this specific example, this means that `Mesh::addChild()` is called, which roughly looks as follows:

```
void Mesh::addChild(NoriObject *obj) {
    switch (obj->getClassType()) {
        case EBSDF:
            if (m_bsdf)
                throw NoriException(
                    "Mesh: multiple BSDFs are not allowed!");
            /// Store pointer to BSDF in local instance
            m_bsdf = static_cast<BSDF *>(obj);
            break;
    // ..(omitted)..
}
```

This function verifies that the nested object is a BSDF, and that no BSDF was specified before; otherwise, it throws an exception of type NoriException.

The following different types of properties can currently be passed to objects within the XML description language:

```
<!-- Basic parameter types -->
<string name="property name" value="arbitrary string"/>
<boolean name="property name" value="true/false"/>
<float name="property name" value="float value"/>
<integer name="property name" value="integer value"/>
<vector name="property name" value="x, y, z"/>
<point name="property name" value="x, y, z"/>
<color name="property name" value="r, g, b"/>
```

```
<!-- Linear transformations use a different syntax -->
<transform name="property name">
    <!-- Any sequence of the following operations: -->
    <translate value="x, y, z"/>
    <scale value="x, y, z"/>
    <rotate axis="x, y, z" angle="deg."/>
    <!-- Useful for cameras and spot lights: -->
    <lookat origin="x,y,z" target="x,y,z" up="x,y,z"/>
</transform>
```

5.2 Visualizing OpenEXR files

For visualizing `.exr` files, we recomend the excellent viewer `TEV`[5], which has support for either Windows, Linux, and Mac OS. However, any other tool of your choice could be used, such as `Adobe Photoshop`, `HDRITools`[6] by Edgar Velázquez-Armendáriz, or `HDRView`[7] by Prof. Wojciech Jarosz. Note that all of them work correctly, but other tools might not really do what one would expect: For example, `Preview.app` on Mac OS for instance tonemaps these files in an awkward and unclear way. If in doubt, you can also use NORI as an OpenEXR viewer: simply run it with an EXR file as parameter, like so:

```
$ ./nori test.exr
```

6 Creating your first Nori classes

In NORI, rendering algorithms are referred to as integrators because they generally solve a numerical integration problem. The remainder of this section explains how to create your first integrators in NORI. We will start by creating a dummy integrator that visualizes the

[5] https://github.com/Tom94/tev
[6] https://bitbucket.org/edgarv/hdritools/downloads/
[7] https://bitbucket.org/wkjarosz/hdrview

surface normals of objects. Then, we will create our first light transport-based integrator, that will allow us to handle **point light sources**.

6.1 NormalIntegrator

We begin by creating a new NORI object subclass in `src/normals.cpp` with the following content:

```
#include <nori/integrator.h>

NORI_NAMESPACE_BEGIN

class NormalIntegrator : public Integrator {
public:
    NormalIntegrator(const PropertyList &props) {
        m_myProperty = props.getString("myProperty");
        std::cout << "Parameter value was : " << m_myProperty << std::endl;
    }

    /// Compute the radiance value for a given ray. Just return green here
    Color3f Li(const Scene *scene, Sampler *sampler, const Ray3f &ray) const {
        return Color3f(0, 1, 0);
    }

    /// Return a human-readable description for debugging purposes
    std::string toString() const {
        return tfm::format(
            "NormalIntegrator[\n"
            "  myProperty = \"%s\"\n"
            "]",
            m_myProperty
        );
    }
protected:
    std::string m_myProperty;
};

NORI_REGISTER_CLASS(NormalIntegrator, "normals");
NORI_NAMESPACE_END
```

This integrator is so far very simple, since only stores a `std::string` for debuggin purposes, and renders all green. You can identify three main elements elements here: First, when creating any NORI object, you will pass to the constructor a list of properties stored in `PropertyList`, which is created by the scene parser. You can search for properties using the functions `PropertyList::getXXX()` (see `include/nori/proplist.h` for all the getters). Then, the most important function is `Li`, that returns the color computed by the integrator (in this case, just green). Finally, the function `toString()` is shared by all NORI objects, and returns a human-readable description of the object. Note that in line 30 we include the `NORI_REGISTER_CLASS` macro, that links the class `NormalIntegrator` with the alias `normals` used in NORI's XML files.

To try out this integrator, we first need to add it to the CMake build system: For this, open `CMakeLists.txt` and look for the command

```
...
add_executable(nori,
```

```
  # Header files
  include/nori/bbox.h
  ...

  # Source code files
  src/bitmap.cpp
  ...
)
```

Add the line `src/normals.cpp` at the end of the source file list as:

```
...
add_executable(nori,
  # Header files
  include/nori/bbox.h
  ...

  # Source code files
  src/bitmap.cpp
  ...
  src/normals.cpp
)
```

You will need to do this for all classes you create in NORI. Note that in Windows you might need to re-run CMake for updating your VS solution. Recompile and if everything goes well, CMake will create an executable named `nori` (or `nori.exe` on Windows) which you can call on the command line.

Finally, create a small test scene with the following content and save it as test.xml:

```
<?xml version="1.0"?>

<scene>
    <integrator type="normals">
        <string name="myProperty" value="Hello!"/>
    </integrator>

    <camera type="perspective"/>
</scene>
```

This file instantiates our integrator and creates the default camera setup. Now, run NORI with this scene by introducing the following command in the console:

```
$ ./nori test.xml
```

Running NORI with this scene causes two things to happen: First, some text output should be visible on the console:

```
Property value was : Hello!

Configuration: Scene[
  integrator = NormalIntegrator[
    myProperty = "Hello!"
  ],
```

```
  sampler = Independent[sampleCount=1]
  camera = PerspectiveCamera[
    cameraToWorld = [1, 0, 0, 0;
                     0, 1, 0, 0;
                     0, 0, 1, 0;
                     0, 0, 0, 1],
    outputSize = [1280, 720],
    fov = 30.000000,
    clip = [0.000100, 10000.000000],
    rfilter = GaussianFilter[radius=2.000000, stddev=0.500000]

  ],
  medium = null,
  envEmitter = null,
  meshes = {
  }
]

Rendering .. done. (took 93.0ms)
Writing a 1280x720 OpenEXR file to "test.exr"
```

The NORI executable echoed the property value we provided, and it printed a brief human-readable summary of the scene, and a window with the rendered image pops up (in this case, a solid green window). Finally, **once the window is closed**, the rendered scene is saved as an OpenEXR file named `test.exr`, and a PNG image named `test.png`. In case you would like to render an image directly, without displaying it on a pop-up window, you can use the command `-b` or `--nogui` when calling NORI, as in the following example:

```
$ ./nori test.xml --nogui
```

Tracing rays

Let's now build a more interesting integrator which traces some rays against the scene geometry. Change the file `normals.cpp` as shown on below:

```
#include <nori/integrator.h>
#include <nori/scene.h>

NORI_NAMESPACE_BEGIN

class NormalIntegrator : public Integrator {
public:
    NormalIntegrator(const PropertyList &props) {
        /* No parameters this time */
    }

    Color3f Li(const Scene *scene, Sampler *sampler, const Ray3f &ray) const {
        /* Find the surface that is visible in the requested direction */
        Intersection its;
        if (!scene->rayIntersect(ray, its))
```

```
            return Color3f(0.0f);

        /* Return the component-wise absolute
           value of the shading normal as a color */
        Normal3f n = its.shFrame.n.cwiseAbs();
        return Color3f(n.x(), n.y(), n.z());
    }

    std::string toString() const {
        return "NormalIntegrator[]";
    }
};

NORI_REGISTER_CLASS(NormalIntegrator, "normals");
NORI_NAMESPACE_END
```

In the code above, we intersect a ray with the scene by calling `scene→rayIntersect()` in line 15. This function returns true or false depending on whether an intersection exists, and if one does, it stores intersection information into the provided `Intersection` record (see Section 5. In the above example we return a color based on the surface normal at the hitpoint, which we compute by extracting the normal from the `Intersection`'s local `Frame` (note that for display, we compute the absolute value of the normal using `Normal::cwiseAbs()`). Also note that no parameter is read from the property list in the constructor, and that `NormalIntegrator::toString()` just displays the name of the integrator.

Invoking NORI on the file `scenes/assignment-1/bunny/bunny-normals.xml` as:

```
$ ./nori scenes/assignment-1/bunny-normals.xml
```

should produce the image shown in Figure 1, a shading normal rendering of the BUNNY scene.

6.2 DirectWhittedIntegrator

So far we have created a very, very simple integrator that simply shows the normals of the object. While simple, that is enough to take a look on some basic operations on integrators such as tracing rays and getting the shading normal of a surface. Now let us get us our hands dirtier by implementing an integrator that actually computes a very simplified light transport model: **direct light from point light sources**. For that, we will need to define a new type of light source (`PointLight`) and a new integrator (`DirectWhittedIntegrator`), while we will use the simple Lambertian shading already implemented in `src/diffuse.cpp`. This first integrator will handle direct illumination with ray-traced shadows from point lights (the kind of scenes that Turner Whitted's ray tracing handled back in 1978).

First of all, let's create two new files in `src/` called `pointlight.cpp` and `direct_whitted.cpp`, and add them to the CMake build system, opening `CMakeLists.txt` and adding two additional lines:

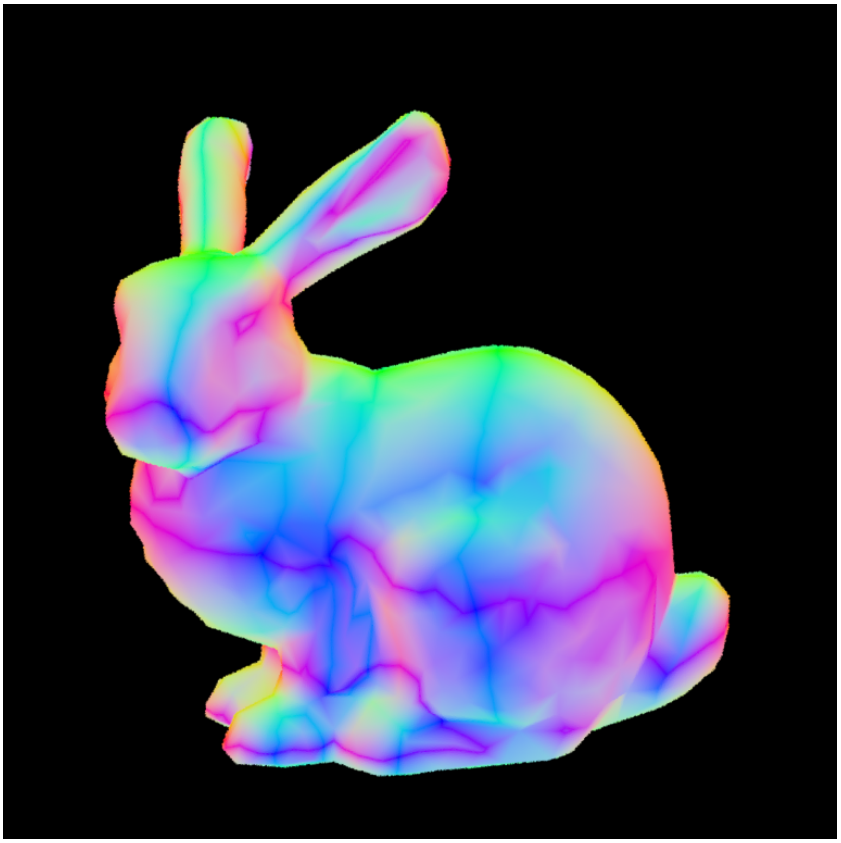

Figure 1: *A shading normal rendering of the* BUNNY *scene.*

```
...
add_executable(nori,
  # Header files
  include/nori/bbox.h
  ...

  # Source code files
  src/bitmap.cpp
  ...
  src/normals.cpp
  src/pointlight.cpp
  src/direct_whitted.cpp
)
```

Re-run CMake to include it in the make system. Now it is time for coding both sources. Let us start with `pointlight.cpp`, which will implement a `PointEmitted`, and therefore will inherit from `Emitter`, as:

```
#include <nori/emitter.h>

NORI_NAMESPACE_BEGIN

class PointEmitter : public Emitter {
public:
  PointEmitter(const PropertyList &props) {
    m_type = EmitterType::EMITTER_POINT;
    m_position = props.getPoint("position", Point3f(0.,100.,0.));
    m_radiance = props.getColor("radiance", Color3f(1.f));
  }

  virtual std::string toString() const {
    return tfm::format(
      "PointEmitter[\n"
      "  position = %s,\n"
      "  radiance = %s,\n"
      "]",
      m_position.toString(),
      m_radiance.toString());
  }

  virtual Color3f eval(const EmitterQueryRecord & lRec) const {
    // This function assumes that a ray have been traced towards
    // the light source. However, since the probability of randomly
    // sampling a point in space is 0, its evaluation returns 0.
    return 0.;
  }
  virtual Color3f sample(EmitterQueryRecord & lRec,
                                const Point2f & sample,
                                float optional_u) const {

    lRec.p = m_position;
    lRec.dist = (lRec.p - lRec.ref).norm();
    lRec.wi = (lRec.p - lRec.ref) / lRec.dist;

    // Note that the pdf should be infinite, but for numerical
    // reasons it is more convenient to just leave as 1
    lRec.pdf = 1.;

    // Note that here it is assumed perfect visibility; this means
    // that visibility should be taken care of in the integrator.
    return m_radiance/(lRec.dist*lRec.dist);
  }

  // Note that the pdf should be infinite, but for numerical reasons
  // it is more convenient to just leave as 1
```

```
  virtual float pdf(const EmitterQueryRecord &lRec) const {
    return 1.;
  }


protected:
  Point3f m_position;
  Color3f m_radiance;
};

NORI_REGISTER_CLASS(PointEmitter, "pointlight")
NORI_NAMESPACE_END
```

Essentially, we have implemented the `PointLight::sample()` function, that fills out an `EmitterQueryRecord` and returns the radiance emitted by the light in point `lRec.p` as a `Color3f`. The remaining functions (`eval()` and `pdf()`) do not have utility for singular light sources, as we will explain later in the course (essentially, a point light is a singularity in space, and therefore cannot be sampled).

Once we have a light source to light the scene, we move to `direct_whitted.cpp`, where we will implement `DirectWhittedIntegrator`, a simple integrator with no parameters nor preprocessing, so that the only function that matters is `Li()`. This is the code for implementing `DirectWhittedIntegrator`. **Note that some parts are missing, as you will have to fill them out:**

```
#include <nori/warp.h>
#include <nori/integrator.h>
#include <nori/scene.h>
#include <nori/emitter.h>
#include <nori/bsdf.h>

NORI_NAMESPACE_BEGIN

class DirectWhittedIntegrator: public Integrator {
public:
  DirectWhittedIntegrator(const PropertyList& props) {
    /* No parameters this time */
  }

  Color3f Li(const Scene* scene, Sampler* sampler, const Ray3f& ray) const {

    Color3f Lo(0.);

    // Find the surface that is visible in the requested direction
    Intersection its;
    if (!scene->rayIntersect(ray, its))
      return scene->getBackground(ray);

    float pdflight;
    EmitterQueryRecord emitterRecord(its.p);

    // Get all lights in the scene
    const std::vector<Emitter*> lights = scene->getLights();

    // Let's iterate over all emitters
    for (unsigned int i = 0; i < lights.size(); ++i)
    {
      const Emitter* em = lights[i];

      // Here we sample the point sources, getting its radiance
      // and direction.
```

```
      Color3f Le = em->sample(emitterRecord, sampler->next2D(), 0.);

      // Here perform a visibility query, to check whether the light
      // source "em" is visible from the intersection point.
      // For that, we create a ray object (shadow ray),
      // and compute the intersection
      /*
       *
       * YOUR CODE HERE
       *
       */

      // Finally, we evaluate the BSDF. For that, we need to build
      // a BSDFQueryRecord from the outgoing direction (the direction
      // of the primary ray, in ray.d), and the incoming direction
      // (the direction to the light source, in emitterRecord.wi).
      // Note that: a) the BSDF assumes directions in the local frame
      // of reference; and b) that both the incoming and outgoing
      // directions are assumed to start from the intersection point.
      BSDFQueryRecord bsdfRecord(its.toLocal(-ray.d),
                                 its.toLocal(emitterRecord.wi), its.uv, ESolidAngle);

      // For each light, we accomulate the incident light times the
      // foreshortening times the BSDF term (i.e. the render equation).
      Lo += Le * its.shFrame.n.dot(emitterRecord.wi) *
                                 its.mesh->getBSDF()->eval(bsdfRecord);
    }

    return Lo;
  }

  std::string toString() const {
    return "Direct Whitted Integrator []";
  }
};

NORI_REGISTER_CLASS(DirectWhittedIntegrator, "direct_whitted");
NORI_NAMESPACE_END
```

What `DirectWhittedIntegrator::Li()` does is to compute the radiance from `Ray ray` by going through all light sources in the scene (lines 27-31), accomulating their contribution according to the rendering equation: First the amount of light from the light source is computed (line 37), as well as the direction from the light source to the illuminated point. Then, **you should modify the code to evaluate whether the point is in shadows or not (lines 39-47).** Finally (lines 49-62), if the point is not in shadow, compute the amount of light reflected by multiplying the incoming radiance in `Le` with the dot product term modeling foreshortening and the evaluation of the surface's BSDF (which you can access by using `its.mesh->getBSDF()`.

Invoking NORI on the file `scenes/assignment-1/serapis-whitted.xml` as:

```
$ ./nori scenes/assignment-1/serapis-whitted.xml
```

should produce the image shown in Figure 2, a direct light rendering of the SERAPIS scene illuminated by a single point light source.

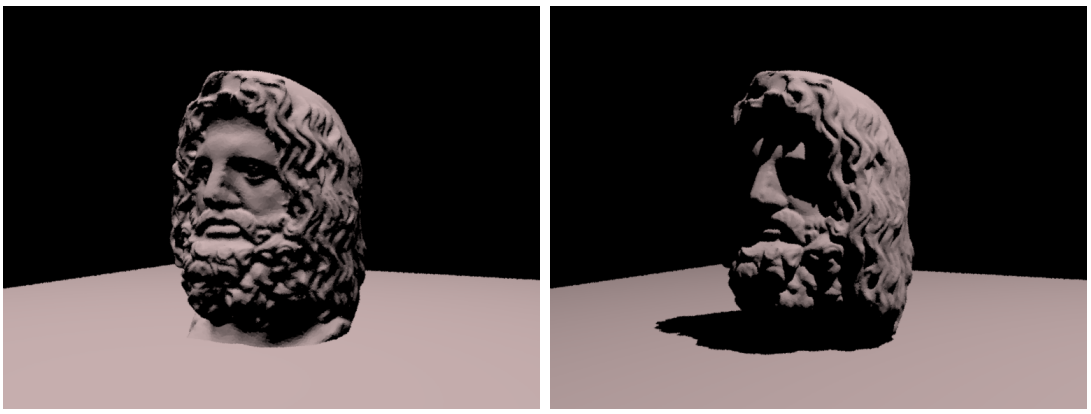

Figure 2: *The* SERAPIS *scene rendered with direct illumination and a single point light source. Left: original code that does not check for shadows. Right: modified code to account for shadows.*

6.3 DepthIntegrator

Lastly, you will code a simple integrator that computes the depth of each point in the scene to the origin of the camera. You implement do this part however you prefer. We recommend creating a new file `src/depth.cpp`, adding it to `CMakeLists.txt`, and modifying `scenes/assignment-1/serapis-whitted.xml` to use your new integrator. Figure 3 shows the expected output: each pixel represents $1/d$, where d is the distance from the camera origin to the first intersection point along the ray.

7 What to submit?

While this assignment is **not evaluable**, to check the completion of the task you should submit the `exr` and `png` image outputs of the `scenes/assignment-1/bunny-normals.xml`, `scenes/assignment-1/serapis-whitted.xml`, and of the modified `serapis-whitted.xml` that uses the depth integrator. Zip all images in a single `zip` file with name "`p1_NIP1_NIP2.zip`", and submitted in Moodle as a response of the assignment, with NIP1 and NIP2 the university ID numbers of each member of the team (remember, max two people per team!).

The **deadline** for this task will be just before the first session of Assignment #2 corresponding to the lab group you are enrolled on:

- **Group 1 (Tuesdays B): October 21, 2024.**
- **Group 2 (Thursdays A): October 16, 2024.**

You will find different submission entries for each group in Moodle. **Make sure you submit**

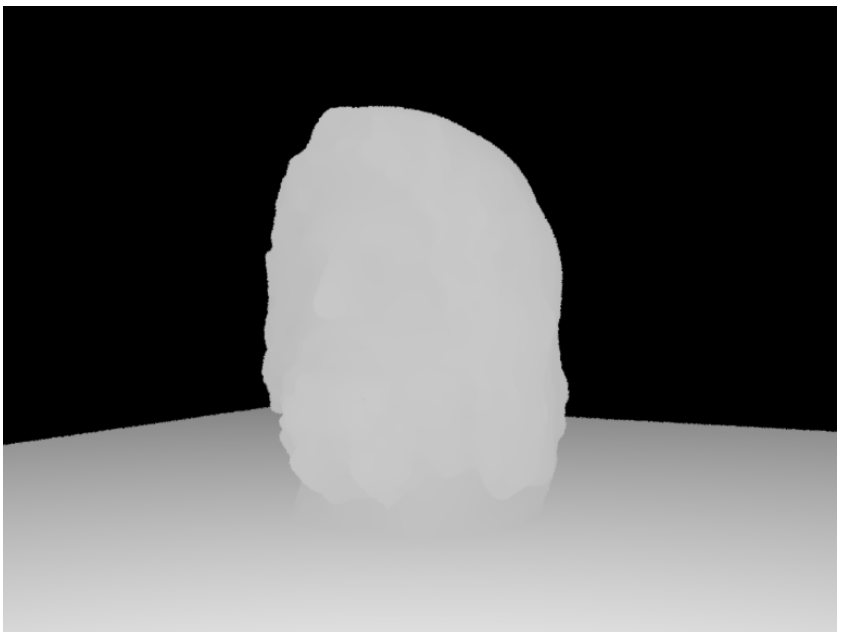

Figure 3: *A render showing depth in the* SERAPIS *scene.*

to the one corresponding to your group.