CSCI 780 – Homework 2: Bidirectional Reflectance Distribution Functions

Out: October 20th (updated Oct. 22nd)
Due: December 1st, noon

Goal

Implement **one** of the following BRDF models:

1. Bagher et al.'s SGD model; or
2. Low et al.'s smooth surface model; or
3. Low et al.'s microfacet model.

You will need to implement one of these models in the Mitsuba rendering system (https://www.mitsuba-renderer.org/), and you will need to demonstrate your implementation on six measured BRDFs from the MERL MIT BRDF database (http://people.csail.mit.edu/wojciech/BRDFDatabase/). In particular the following six materials:

1. Aluminum-Bronze
2. Blue-Acrylic
3. Blue-Rubber
4. Chrome
5. Gold-Metallic-Paint2
6. Gold-Paint

Rendering: Mitsuba

You can find a slightly modified version of Mitsuba on Blackboard (mitsuba.tgz) that compiles on the machines in the department. To install:

```
tar xzvf mitsuba.tgz
cd mitsuba
cmake .
make
```

If everything goes well, 'cmake .' will produce the correct makefiles. It will say that a few (non-critical) libraries are not present, but all required libraries should be found without issue. Note: compilation will take a while. In case for some reason you'd like to use the latest version directly from the mitsuba website, the following changes were applied to make it compile on the departmental machines: 1) added the 'Eigen3' library in mitsuba's include directory, 2) added './include' to the search path in 'data/cmake/FindEigen.cmake', and 3) removed 'NO_DEFAULT_PATH' from the include and lib search

path in 'data/cmake/FindXerces.cmake'. In case you like to install mitsuba on your own desktop, it is easiest to employ a trail-and-error strategy where you run 'cmake .', see what it complained about, fix it, and try again (possible removing CMakeCache before rerunning cmake).

Mistuba is a rather extensive global illumination package. For the purpose of this homework, you do **NOT** need to understand the complete package. However, the manual (https://www.mitsuba-renderer.org/releases/current/documentation_lowres.pdf) is a good source of (non-coding) information in case you need more information.

In order to aid in developing and comparing your results, two new BRDF implementations (Cook-Torrance and the direct rendering of the measured MERL BRDFs) are included on Blackboard (example_bsdfs.tgz). To install:

1. Unpack the content example_bsdfs.tgz (3 files) in the 'src/bsdfs' subdirectory in your mitsuba install directory.
2. Next, we will need to tell Mistuba that there are two new BRDFs. For this add the following lines to the end of 'CMakeLists.txt' in the 'src/bsdfs'subdirectory:

```
add_bsdf(merl   MERL.cpp)
add_bsdf(cooktorrance   cooktorrance.cpp)
```

3. Rerun 'cmake .' in the main mitsuba directory, and compile (it should only recompile the new files).

You will need to follow a similar process if you want to add your own BRDF model. I would suggest you copy 'cooktorrance.cpp' to a new BRDF and modify it. In the next section I'll describe the methods that each BRDF should implement.

On Blackboard you will also find all necessary files for rendering the six materials with each of the two new models. For this, download 'scene.tgz', and unpack it in your main mitsuba directory. This will produce a subdirectory 'scenes' which contains the MERL MIT BRDFs (in 'scenes/merlmit'), a light probe ('grace.exr'), and 6 scene files 'sphere_XX_YYY.xml' where XX is either 'ct' (Cook-Torrance BRDF) or 'merl' (directly using the MERL data), and where YYY is the material name. To render (assuming you successfully completed the above steps), just run in your mitsuba install directory: './binaries/mitsuba ./scenes/sphere_XX_YYY.xml' (with XX and YYY set to the desired combination). It will take around 30 seconds to 1 minute to produce an EXR image with the same name (in the 'scenes' subdirectory).

Scene File Description

Mitsuba uses an XML format to describe the scene. This section will only focus on the relevant portions for this homework assignment, in particular the specification of the material properties. A full description of the scene file format can be found in the mitsuba manual (https://www.mitsuba-renderer.org/releases/current/documentation_lowres.pdf). As a first example, we look "sphere_ct_alum_broze.xml". The material is specified in the following lines:

```
<bsdf type="cooktorrance" id="alum-bronze-cook-torrance">
    <rgb name="diffuseReflectance" value="0.0886, 0.0602, 0.0299"/>
    <rgb name="specularReflectance" value="0.118, 0.0698, 0.0371"/>
    <float name="F0" value="0.203"/>
    <float name="roughness" value="0.0367"/>
</bsdf>
```

This states that a 'bsdf' of type 'cooktorrance' is defined, and that this particular instance can be referred to by the (user selected) identification 'alum-bronze-cook-torrance'. Thus anytime we want to use this particular BRDF instance, we can just use this id. For example:

```
<shape type="sphere">
    <float name="radius" value="1"/>
    <ref id="alum-bronze-cook-torrance"/>
</shape>
```

This creates a sphere with radius 1 (located at the origin) and with the material set to our previously defined BRDF. Note, we can replace the line '<ref id=…./>' also by directly copying the above definition of the BRDF instead.

The general structure of a BRDF specification is: <bsdf type='…'/> parameters </bsdf>, where the exact nature of the parameters are model-dependent. The Cook-Torrance BRDF is defined in the file 'src/bsdfs/cooktorrance.cpp'. **Note:** the type name is actually defined in the CMakeList.txt in 'src/bsfds' by the line: 'add_bsdf(cooktorrance cooktorrance.cpp)' that binds the type name 'cooktorrance' to 'cooktorrance.cpp'.

In 'cooktorrance.cpp' a class 'CookTorrance' (a specialization of 'BSDF') is defined. This implementation follows the definition of the Cook-Torrance BRDF as specified in the introduction of the supplemental material from Ngan et al. 2005 (available on BlackBoard). There are two constructors:

```
CookTorrance(const Properties &props) : BSDF(props) {
  m_diffuseReflectance = props.getSpectrum("diffuseReflectance", Spectrum(0.5f));
  m_specularReflectance = props.getSpectrum("specularReflectance", Spectrum(0.2f));
  m_roughness = props.getFloat("roughness", 0.1f);
  m_F0 = props.getFloat("F0", 0.1f);
}
```

and:

```
CookTorrance(Stream *stream, InstanceManager *manager) : BSDF(stream, manager) {
  m_diffuseReflectance = Spectrum(stream);
  m_specularReflectance = Spectrum(stream);
  m_roughness = stream->readFloat();
  m_F0 = stream->readFloat();
```

```
    configure();
}
```

The first is the default constructor, which assigns in this case 50% grey to the diffuse albedo (called 'diffuseReflectance' in the XML file, and stored in the class attribute 'm_diffuseReflectance'), 20% grey to the specular albedo ('specularReflectance' and 'm_spectularReflectance' respectively), a roughness of 0.1 ('roughness' and 'm_roughness' respectively), and 0.1 to the total Fresnel reflectance at normal incidence ('F0' and 'm_F0' respectively). Note, that the first two attributes are defined as a spectrum, and the last two as a float. This translates itself to the XML format as defining the scalar floats as: <float name="F0" value="0.1"/> for example. It is a little bit more complicated for a spectrum (mitsuba uses a multispectral representation internally instead of the typical RGB representation). Since, we are only interested in RGB colors, we will specify a color in XML as follows: '<rgb name="diffuseReflectance" value="0.5, 0.5, 0.5"/>' where the RGB color is passed as a comma separated triplet with the values in the 0 to 1 range. Mitsuba will internally convert the RGB color to a spectrum.

The second constructor is responsible for reading the values from the XML file. The XML file is passed in the form of a stream to the constructor. Most variable types can be read by passing the stream to the type constructor, e.g., 'Spectrum(stream)'. Default types, such as float, are handled differently, and we have to explicitly call a read function on the stream: 'stream->readFloat()'.

In the case of 'alum-bronze', we directly copied the values from the supplemental material from Ngan et al. 2005 as defined on p13: (d_r, d_g, d_b) -> diffuseReflectance, (s_r, s_g, s_b) -> specularReflectance, p0 -> F0, p1 -> roughness.

As a second example, let's look at 'sphere_merl_alum_bronze.xml' which is the same scene as above, except that it uses the measured BRDF values from the MERL database directly:

```
<bsdf type="cooktorrance" id="alum-bronze-cook-torrance">
    <rgb name="diffuseReflectance" value="0.0886, 0.0602, 0.0299"/>
    <rgb name="specularReflectance" value="0.118, 0.0698, 0.0371"/>
    <float name="F0" value="0.203"/>
    <float name="roughness" value="0.0367"/>
</bsdf>

<bsdf type="merl" id="alum-bronze">
    <string name="binary" value="./merlmit/alum-bronze.binary"/>
    <ref id="alum-bronze-cook-torrance"/>
</bsdf>
```

Note that the beginning is exactly the same as before: a Cook-Torrance BRDF definition. The second part defines the MERL BRDF (named 'alum-bronze'), which loads the 'binary' from './merlmit/alum-bronze.binary' (the path is relative to the location of the XML file), and it takes as a second parameter the Cook-Torrance BRDF. The reason for this second (nested) BRDF is to support efficient importance sampling. The measured BRDF value is used for evaluating the BRDF, and the second BRDF is **only** used

for importance sampling (selecting a direction) -- **the second BRDF is never evaluated**, and you can actually specify any BRDF here (it only affects the convergence rate of the rendering – the closer the importance sampling BRDF is in shape to the measured MERL BRDF, the faster the convergence). This is specified in the constructors in 'MERL.cpp' as:

```
MERL(const Properties &props) : BSDF(props) {
  m_importance = NULL;
  m_name = props.getString("binary", "blue-acrylic.binary");
}

MERL(Stream *stream, InstanceManager *manager) : BSDF(stream, manager) {
      m_name = stream->readString();
      m_importance = static_cast<BSDF *>(manager->getInstance(stream));
      m_importance->incRef();
      configure();
}
```

Note that the default constructor sets m_importance (the importance sampling guide) to NULL, and hence you cannot render this 'default' BRDF'. The stream constructor, reads a string (the filename of the MERL binary' and stores it in 'm_name', and then gets an instance of the importance sampling BRDF. The latter is achieved by requesting an instance from the InstanceManager based on the XML stream data (manager->getInstance(stream)), subsequently casting it to a pointer of the right class (BSDF*), and finally, notifying the system that an additional instance is used (increasing the reference count).

Note: you should always call 'configure()' after the construction of the BSDF. See below for more information regarding this method.

Mitsuba BRDF – Methods

To define your own BRDF you will need to implement a number of methods:

- **Default Constructor**: <classname>(const Properties &props) : BSDF(props) { … }. The default constructor takes a single parameter of properties filled in by the system. You simply pass these properties to the parent class BSDF, and set the class attributes to default values. This constructor is called when an instance is created and the values will be used when a user does not assign a specific value to a class attribute in the scene file.
- **Stream constructor**: <classname>(Stream *stream, InstanceManager *manager) : BSDF(stream, manager) { … }. Receives two parameters, the XML stream, and the XML instance manager. Unless you use nested objects (which is normally not needed for this homework) you will not have to deal with the instance manager. Essentially, you will need to extract each of the BRDF parameters specified in the XML scene file and store them in a class attribute. At the end you should call the 'configure' method.

- **void configure(void)** configures the BSDF. You can think of the configure method as an extended constructor. Generally, more complex computations are executed here instead of in the constructor. For example, for the MERL BRDF, this is where the MERL binary is loaded. Note, unlike the constructor, the configure method can fail, and a message is send to the log (e.g., 'Log(EError, "Unable to find \"%s\".", m_name.c_str());'). You **must** also start each configure method with the following lines:

```
m_components.clear();
m_components.push_back(EGlossyReflection | EFrontSide );
m_components.push_back(EDiffuseReflection | EFrontSide );
m_usesRayDifferentials = false;
```

  which sets all the reflection components to only affect the front side of the material.
- **void serialize(Stream *stream, InstanceManager *manager) const**: does the inverse of the stream constructor: it converts the BRDF instance to a XML object. You **must** start each of your serialize implementations with 'BSDF::serialize(stream, manager);'. Also, this method should serialize your objects in exactly the reverse order as the stream constructor (compare the order of 'diffuseReflectance', 'specularReflectance', 'F0' and 'roughness' in 'cooktorrance.cpp' in the stream constructor and the serialize method).
- **std::string toString() const**: convert the BRDF instance to a human readable form. Refer to the toString implementation in 'cooktorrance.cpp' and 'MERL.cpp'. This method is very useful when debugging.
- **Float getRoughness(const Intersection &its, int component)**: should return a rough estimate of the roughness. Some rendering algorithms will use this to decide how to handle a material. If unsure, you can omit this function.
- **Spectrum eval(const BSDFSamplingRecord &bRec, EMeasure measure) const**: evaluates the BRDF for a given incoming and outgoing direction (bRec.wi and bRec.wo respectively). Note that this method return the **BRDF times the foreshortening** (= Frame::cosTheta(bRec.wo)). The Cook-Torrance implementation is a good guide.

```
/* sanity check */
if(measure != ESolidAngle || Frame::cosTheta(bRec.wi) <= 0 ||Frame::cosTheta(bRec.wo) <= 0)
    return Spectrum(0.0f);

/* which components to eval */
bool hasSpecular = (bRec.typeMask & EGlossyReflection)
                && (bRec.component == -1 || bRec.component == 0);
bool hasDiffuse  = (bRec.typeMask & EDiffuseReflection)
                && (bRec.component == -1 || bRec.component == 1);

/* evaluate */
Spectrum result(0.0f);
if (hasSpecular) { result += … }  // specular * foreshortning
```

```
if (hasDiffuse) { result += … } // diffue * foreshortening

// Done.
return result;
```

The first part check if the incident and outgoing directions lie above the horizon – you cannot reflect light from below the surface. If this fails, return 'black'. The second part, checks which part of the BRDF to evaluate: diffuse or specular. It sets the 'hasSpecular' and 'hasDiffuse' booleans. Next, we evaluate the specular component (times forshortning) and **add** it to result. Finally, we evaluate the diffuse component (times foreshortening) and **add** it to the result. Confusingly, mitsuba uses the reverse convention regarding what is incident and outgoing directions of what one would expect: **bRec.wo is the light direction,** and **bRec.wi is the view direction.**

- **Float pdf(const BSDFSamplingRecord &bRec, EMeasure measure) const**: returns the PDF of sampling the direction bRec.wi from a given direction bRec.wo. You do not need to sample the directions, just return the PDF of the sampling.
- **Spectrum sample(BSDFSamplingRecord &bRec, Float &pdf, const Point2 &_sample) const**: given an incoming direction 'bRec.wi', and a uniform random (2D) variable 'sample', compute the 'pdf' and sample an outgoing direction 'bRec.wo'. Finally, this method also return an evaluation of the BRDF **divided by the 'pdf'**. The reason for putting everything together in one function, is so that you can reuse computations. However, I recommend that you forego efficiency and produce clean and readable code. Hence, I recommend that you (as in 'cooktorrance.cpp' end this method by directly calling the 'eval' method for evaluating the BRDF: 'return eval(bRec, ESolidAngle) / pdf;'. It is recommended to start your sample method by checking whether a diffuse and/or a specular component (or whether it is requested). Next, stochastically select the component (diffuse or specular) to sample (probability of selecting diffuse and specular should be proportional to their respective albedos), and then based on this selection, sample the BRDF according to the selected component. Essentially, you can copy and adapt most of the Cook-Torrance sample method, and only replace the code inside the 'if(choseSpecular) { …}' and 'if(choseDiffuse) {…}' statements.
- **Spectrum sample(BSDFSamplingRecord &bRec, const Point2 &sample) const**: is the same as the above, except that the PDF is not returned. It is easiest to implement this as in cooktorrance.cpp, and just create a temporary PDF variable, and return the result from the above method.

Methods not discussed in this overview, can be copied from their default implementations as in 'cooktorrance.cpp' and 'MERL.cpp'. For more information please check "include/mitsuba/render/bsdf.h" for a more implementation driven explanation of each methods.

Grading

The grading of the homework will be as follows:

- 5% whether or not your code compiles. I will not correct your code or make alterations, so check that it compiles on the departmental machines.
- 5% on the report.
- 30% on the correctness of the results. You will need to submit a rendering (EXR format) of each of the 6 materials at a resolution of 512x512, rendered with 256 samples per pixel using the same lighting and shape configuration as the examples scenes provided with the assignment.
- 10% on the implementation of the constructors, configure, serialize, and toString methods.
- 20% on the correctness of the implementation of the evaluation.
- 20% on the correctness of the importance sampling (sample).
- 10% on the correctness of the 'pdf' method.
- 30% bonus on the 2nd BRDF (following the same grading as above but then each percentage divided by 3; not including report and compilation). This part is a **bonus and is optional**.
- 15% bonus on the 3th BRDF. This part is a **bonus and is optional**.

Debugging Tips

- Implement the default constructor first, and assign some realistic values (e.g., taken from the supplemental material) for a specific material (e.g., alum-bronze or blue-acrylic). Ideally, your rendering of your implementation will look very similar to the ground truth MERL rendering.
- Focus initially on implementing the 'eval' method. Temporarily copy importance sampling from 'diffuse.cpp' to start with. This is a very non-optimal sampling, but correct, strategy and your will get very noisy results, but if you increase the number of samples, you should get a converged correct result. Only start implementing the importance sampling when your rendering converges to the correct solution using the diffuse placeholder sampling methods (i.e., brightness should be about similar to the ground truth MERL rendering, the image should not get brighter/darker when adding samples, etc…).
- You can speed up debugging by setting a lower resolution (in the 'film' part of the XML file) or a lower sampling rate (in the 'ldsampler' part of the XML file).
- You should always do a final debugging check where you increase the sample rate to a large number (e.g., 10000) to see if the method converges and does not produce odd results (e.g., spikes, not-a-number pixel values, etc…).
- While a visual comparison between the ground truth MERL rendering and your result is a good start, it can also hide differences in brightness. Therefore, it is also instructive to look of the difference and ratio of these images (you can use the library provided for homework 1 to compute such images.
- Mitsuba has by default assert and cerr disabled. You can output information to the log instead. Use EError to report an error (and abort) or EWarn to report a warning (and continue): Log(EWarn, "Oops: warning=%d", variableNames); The string formatting is similar as that of

sprintf: %f for floating points, %d for integers, and %s for strings. If you want to output a Spectrum (or any other class instance), use: Log(EWarn, "Spectrum=%s", spectrumVariable.toString().c_str()); In this case we convert the 'spectrumVariable' (or whatever variable you want to print) to a string using the toString method, and then convert the string a C-string (c_str) suitable for sprintf-like printing.

- You can do a statistical test of your BRDF sampler using the provided test program: test_chisquare.cpp (in src/test_chisquare.cpp).
- Be mindful of numerical issues (such as division by zero).

Submission Details

The following **three files** should be emailed to "ppeers@cs.wm.edu" by **December 1st, noon:**

- A report (max 1 page **PDF**) named **<firstname>-<lastname>.pdf** that contains the following information:
  - **Time required to finish the core requirements this homework.**
  - A list of features that work, and a list of features that don't work.
  - A description of difficulties encountered and how you solved them.
- An archive of the renderings (**<firstname>-<lastname>-results.tgz**) of each material and each BRDF model you implemented **as well as the XML files used to generate the results**. The renderings should be of the same scene as for the example BRDFs (a sphere lit by the Grace Cathedral light probe), at a resolution of 512x512 with 256 samples per pixel (essentially, copy the provided example xml scene files, and only replace the material assignment to the sphere with one of your own model). On Blackboard you can find the supplementary documents of Low et al. and Bahger et al. which contains the relevant fitted model parameters for each material in the MERL MIT database (you will need to copy the relevant parameters to the xml scene file):
  - cd mitsuba/scenes
  - **tar czvf <firstname>-<lastname>-results.tgz sphere_<XX>_*.exr sphere_<XX>_*.xml**

  Please ensure all the relevant EXR files are included, and submit this TGZ archive with your report.
- An archive of your code (**<firstname>-<lastname>-code.tgz**):
  - cd mitsuba/src/bsdfs
  - **tar czvf <firstname>-<lastname>-code.tgz <brdf_name>.cpp CMakeList.txt**

  If you want to add other BRDFs, you will need to add them to the file list at the end of the last line above. You cannot added easily to the same tgz archive, so it you need to list all files at once you want to have in the archive. Please ensure that the CMakeLists.txt of the src/bsdfs directory is included, as well as all the code not part of the original code archives (mitsuba.tgz and example_bsdfs.tgz).

Remarks

- You are not allowed to use external libraries, except whatever is provided in the standard C++ libraries, Boost, and what Mitsuba offers.
- Late policy: score -= ceil(hours_late / 24) * 10. Negative scores are possible. An extension needs to be requested at least 48 hours before the deadline. Each extension request needs to clearly justify why the extension is needed.