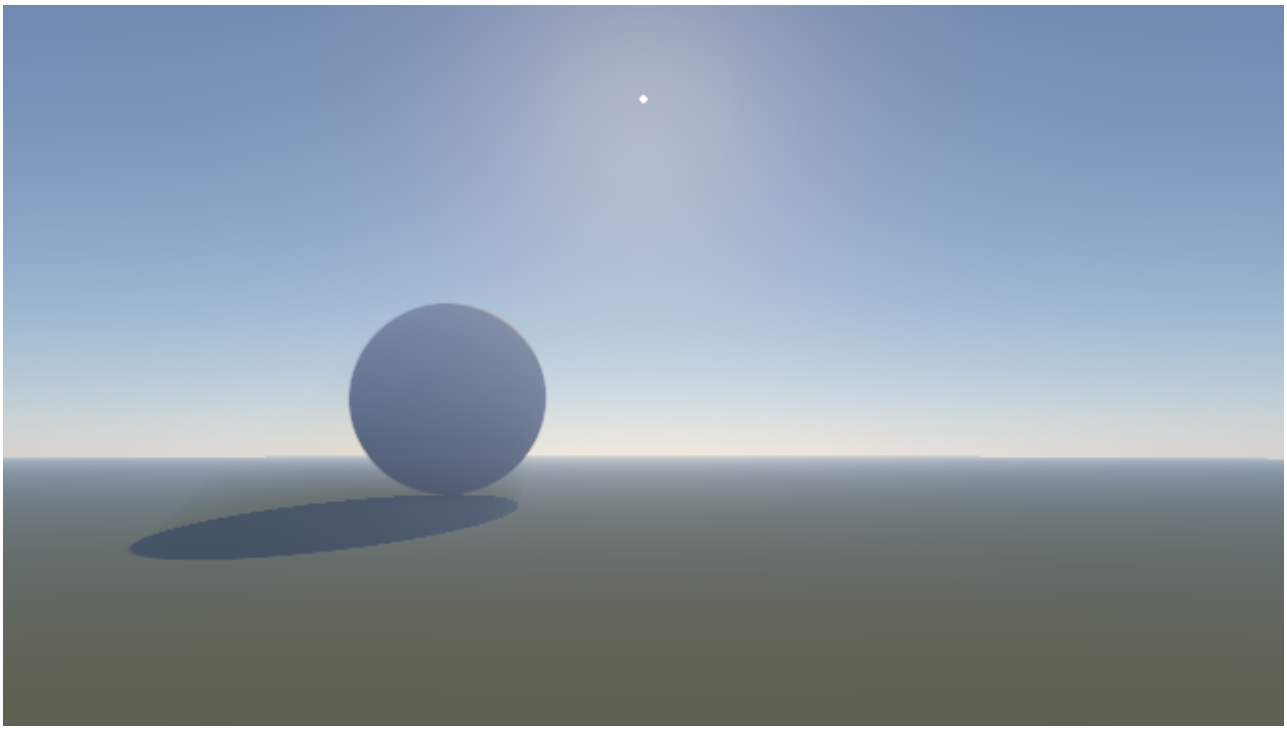
Precomputed Atmospheric Scattering: a New Implementation

Eric Bruneton, 2017



Introduction

This document presents a new implementation of our Precomputed Atmospheric Scattering paper. This new implementation is motivated by the fact that the original implementation:

- has almost no comments and no documentation, and as a result is difficult to understand and to reuse,
- has absolutely no tests, despite the high risk of implementation errors due to the complexity of the atmospheric scattering equations,
- contains ad-hoc constants in its texture coordinates mapping functions which are adapted to the Earth case, but cannot be reused for other planets,
- provides only one of the two options presented in the paper to store the single Mie scattering components (i.e. store the 3 components, or store only one and reconstruct the others with an approximation),
- does not implement the light shaft algorithm presented in the paper,

- uses an extra-terrestrial solar spectrum independent of the wavelength (with an arbitrary and completely unphysical value "100") and displays the radiance values directly instead of converting them first to luminance values (via the CIE color matching functions).

To address these concerns, our new implementation:

- uses more descriptive function and variable names, and adds extensive comments and documentation.
- uses static type checking to verify the dimensional homogeneity of all the expressions, and uses unit tests to check more complex constraints,
- uses slightly improved texture coordinates mapping functions which, in particular, no longer use ad-hoc constants,
- provides the two options presented in the paper to store the single Mie scattering components (which are then compared in our tests),
- partially implement the light shaft algorithm presented in the paper (it implements Eqs. 17 and 18, but not the shadow volume algorithm),
- uses a configurable extra-terrestrial solar spectrum, and either
  - converts the spectral radiance values to RGB luminance values as described in A Qualitative and Quantitative Evaluation of 8 Clear Sky Models (section 14.3),
  - or precomputes luminance values instead of spectral radiance values, as described in Real-time Spectral Scattering in Large-scale Natural Participating Media (section 4.4). The precomputation phase is then slower than with the above option, but uses the same amount of GPU memory.

  This gives almost the same results as with a full spectral rendering method, at a fraction of the cost (we check this by comparing the GPU results against full spectral CPU renderings).

In addition, the new implementation adds support for the ozone layer, and for custom density profiles for air molecules and aerosols.

The sections below explain how this new implementation can be used, present its structure and its documentation and give more details about its tests.

Usage

Our new implementation can be used in C++ / OpenGL applications as explained in model.h, and as demonstrated in the demo in `atmosphere/demo`. To run this demo, simply type `make demo` in the main directory. A WebGL2 version of this demo is also available online.

The default settings of this demo use the real solar spectrum, with an ozone layer. To simulate the settings of the original implementation, set the solar spectrum to "constant", and turn off the ozone layer.

Structure

The source code is organized as follows:

- `atmosphere/`
  - `demo/`
    - `...`

    - reference/
        - ...
    - constants.h
    - definitions.glsl
    - functions.glsl
    - model.h
    - model.cc

The most important files are the 5 files in the `atmosphere` directory. They contain the GLSL shaders that implement our atmosphere model, and provide a C++ API to precompute the atmosphere textures and to use them in an OpenGL application. This code does not depend on the content of the other directories, and is the only piece which is needed in order to use our atmosphere model on GPU.

The other directories provide examples and tests:

- The `atmosphere/demo` directory shows how the API provided in `atmosphere` can be used in practice, using a small C++/OpenGL demo application. A WebGL2 version of this demo is also available, in the `webgl` subdirectory.
- The `atmosphere/reference` directory provides a way to execute our GLSL code on CPU. Its main purpose is to provide unit tests for the GLSL shaders, and to statically check the dimensional homogeneity of all the expressions. This process is explained in more details in the Tests section. This code is also used to compute reference images on CPU using full spectral rendering, in order to evaluate the accuracy of the approximate "radiance to RGB luminance" conversion performed by the GPU shaders. It depends on external libraries such as dimensional_types (to check the dimensional homogeneity) and minpng.

Documentation

The documentation consists of a set of web pages, generated from the extensive comments in each source code file:

- atmosphere
    - demo
        - demo.h
        - demo.cc
        - demo.glsl
        - demo_main.cc
        - webgl
            - demo.js
            - precompute.cc
    - reference
        - definitions.h
        - functions.h
        - functions.cc
        - functions_test.cc
        - model.h
        - model.cc
        - model_test.cc
        - model_test.glsl
    - constants.h
    - definitions.glsl
    - functions.glsl
    - model.h
    - model.cc

Tests

To reduce the risk of implementation errors, two kinds of verifications are performed:

- the dimensional homogeneity is checked at compile time, via static type checking,
- the behavior of each function is checked at runtime, via unit tests.

The main issue to implement this is that a GLSL compiler cannot check the dimensional homogeneity, unlike a C++ compiler (see for instance Boost.Units). Our solution to this problem is to write our GLSL code in such a way that it can be compiled both by a GLSL compiler and by a C++ compiler. For this:

- we use macros to hide the few syntactic differences between GLSL and C++. For instance, we define `OUT(x)` as `out x` in GLSL, and as `x&` in C++, and declare output variables as `OUT(SomeType) someName` in our shaders.
- we define the physical types, such as length or power, in a separate file, which we provide in two versions:
  - the GLSL version defines the physical types as aliases of predefined types, such as `float`,
  - the C++ version defines the physical types based on dimensional_types abstractions, which are designed to produce compile errors when attempting to add, subtract or compare expressions with different physical dimensions.
- we use the predefined GLSL variables such as `gl_FragCoord` only in the `main` functions, which we reduce to the minimum (e.g. `main() { gl_FragColor = Main(gl_FragCoord); }`) and exclude from the C++ compilation.

Thanks to this double GLSL and C++ compilation, the unit tests for the GLSL code can then be implemented either in GLSL or in C++. We chose C++ because it is much more practical. Indeed, a C++ unit test does not need to send data to the GPU and to read back the test result, unlike a GLSL unit test.