

# PARALLEL\_RAYTRACE(1) Rayverb User Manual

Reuben Thomas

March 9, 2015

## NAME

`parallel_raytrace` - fast generator of raytraced impulse responses

## SYNOPSIS

`parallel_raytrace` [configuration-file (.json)] [3D-object-file] [material-file (.json)] [output-file (.aiff)]

## DESCRIPTION

### Overview

`Parallel_raytrace` generates impulse responses of 3D models, using raytracing techniques.

The program takes three input files. Descriptions of these are given below, with valid fields and examples under ‘FILE FORMATS’:

The first, the configuration-file, provides all the vital information, such as the positions of sources and microphones, and speaker configurations, along with any optional flags. This file should be a valid JSON-formatted document. Using JSON allows for passing structured data to the program, which is necessary for defining speaker and hrtf configurations.

The second, the 3D object file, is the model that will actually be traced. `Parallel_raytrace` uses the Assimp library for object input, and therefore supports a variety of different 3D formats, including Collada (.dae), Blender 3D (.blend), 3ds Max 3DS (.3ds), and Wavefront Object (.obj). A full list is available at [the Assimp website](#).

Thirdly, the material file defines the specular and diffuse coefficients of the various materials in the scene. Each material in the 3D model should have an entry with the same name in the material file. For the .obj format this is easy, as all the model’s materials are saved to a separate .mtl file. You can just check this .mtl for the material names, and add appropriate materials to the material file.

Finally, the output file is the impulse response file that will be written. At the moment, only .aiff formats are supported, in 16 or 24 bits, at various samplersates.

## Algorithm Description

The algorithm takes advantage of the ‘embarrassing parallelism’ of raytracing by carrying out the main raytrace on the GPU rather than the CPU.

Each ray is shot in a straight line, in a random direction, from the `source_position`. Each ray starts with a volume of ‘1’ in all 8 frequency bands. When a ray intersects with a piece of geometry (or *primitive*) in the scene, the material of that primitive is checked. A new volume is calculated by multiplying the current volume of the ray by the specular coefficients of the material. Then, a new ray is cast back from the intersection point to the `mic_position`. If this new ray intersects with the `mic_position`, then the mic is ‘visible’ from the intersection point, and a diffuse contribution is added to the output. The volume of this diffuse contribution is calculated by multiplying the ray volume by the diffuse coefficients of the material, and then correcting for air attenuation (that is, energy lost to the air as the ray travels through it) and angle of reflection (direct reflections contribute more than reflections in other directions). The contribution time is calculated by dividing the total distance travelled by the ray by the speed of sound. Then, a new ray is cast, perfectly reflected from the intersection point. The process is repeated with this new ray. In this way, the process continues until the maximum number of reflections, `reflections`, has been reached.

For early reflections, an additional method is used to calculate contributions. When a ray intersects with a primitive, the microphone position is reflected in all the previous primitives that the ray has struck, producing a ‘microphone image’. Then, a ray is traced from the source to this new microphone image. If this new ray intersects with all the intermediate primitives, then there is a direct reflection pattern between the source and the microphone, and a contribution is added for this pattern. Because multiple rays may discover the same direct reflection patterns, duplicates are filtered out in a post-processing step after the raytrace.

The completed raytrace produces a collection of ‘Impulses’, each of which has an 8-band volume, a position, and a time. These impulses are attenuated depending on their direction from the microphone, using either polar-pattern or HRTF coefficients. After attenuation, each band is filtered, and then the bands are summed together to produce a single full-spectrum response. This response can optionally be normalized, volume-scaled, and trimmed. Then it is written to file.

## FILE FORMATS

### Configuration-file

This file has a set of required fields, and a set of optional fields. This file should contain a single JSON object.

The required fields are as follows:

- *rays* - The number of rays that will be traced. The bigger the number, the better the approximation. Around 50000 should work well.
- *reflections* - The maximum number of times each ray can be reflected from a surface. The bigger the number, the longer the reverb tail you’ll be able to emulate. Around 100 reflections should be sufficient for the majority of models.
- *sample\_rate* - The sampling rate of the produced file. 44100 or 48000 should be sufficient in the vast majority of cases.
- *bit\_depth* - The bit depth/dynamic range of the output. Valid values are 16 or 24.
- *source\_position* - The position of the sound source in 3D space. This should be a JSON array [x, y, z], specifying the absolute 3D coordinates of the source in the scene.

- *mic\_position* - The position of the microphone in 3D space. Similar to the *source\_position*, this should be a JSON array [x, y, z].
- *attenuation\_model* - The postprocessing stage that converts the raytrace into multichannel audio. This should be specified as a JSON object with a single key, either **speakers** or **hrtf**.
  - If the key is **speakers**, the value for that key should be a list of speaker definitions. Each speaker definition is a JSON object with two keys, **direction** and **shape**. **direction** corresponds to the direction in which the speaker's polar pattern is pointing, while **shape** corresponds to the shape of the polar pattern. A **shape** of 0.0 corresponds to an omnidirectional pickup, while a shape of 1.0 corresponds to a bipolar pickup. Cardioid is at 0.5.
  - If the key is **hrtf**, the value for that key should be a single object with two keys, **facing** and **up**. Each of these are 3D vectors - **facing** is the direction in which the virtual nose is pointing, and **up** is a vector through the top of the virtual head.

In addition, there are a variety of optional fields:

- *filter* - The filtering method that will be used when downmixing multiband impulse-responses into a single response. Valid values are **sinc**, **onepass**, **twopass**, and **linkwitz\_riley**, corresponding to windowed-sinc, single-pass biquad, two-pass biquad, and linear-phase linkwitz-riley filtering.
- *hipass* - Whether or not to apply a windowed-sinc hipass at 10Hz on the final response. This is mainly useful for removing DC bias.
- *normalize* - Whether or not to normalize the output. Normally, you'll want normalization (so there's no clipping/distortion), but if you're tracing lots of matched impulses of one room you might want them all at the same relative volume, in which case you should set this to **false**. If you disable normalization, you should also set *volume\_scale* low to avoid clipping.
- *volume\_scale* - A global multiplier coefficient. Useful if you don't want normalized responses, but instead want several responses at the same relative level. This value should be in the range (0, 1). You might need to experiment to find a suitable value.
- *trim\_predelay* - Removes predelay from the impulse. For most musical applications, you'll want to keep this enabled, so that your musical material isn't delayed. If you're doing auralization or room-modelling, you might want to disable it.
- *remove\_direct* - Removes the direct source->mic impulse contribution.
- *trim\_tail* - Traces can have very long, nearly inaudible reverb tails. Enable this to trim the quiet reverb tail.
- *output\_mode* - Whether to output diffuse contributions, image-source contributions, or both. You probably want both, but the other modes may be useful for diagnostics. Valid values are **all**, **image\_only**, and **diffuse\_only**.
- *verbose* - If enabled, the program will print additional diagnostic information, such as the model materials found, and OpenCL build information, to stderr.

An example configuration file is shown below:

```
{
  "source_position": [0, 1, 0],
  "mic_position": [0, 1, 2],
  "rays": 50000,
  "reflections": 128,
```

```

"sample_rate": 44100,
"bit_depth": 16,
"attenuation_model":
{
  "speakers":
    [
      {"direction": [-1, 0, -1], "shape": 0.5}
      , {"direction": [ 1, 0, -1], "shape": 0.5}
    ]
  },
"filter": "twopass",
"hipass": true,
"trim_predelay": true,
"trim_tail": true,
"output_mode": "all"
}

```

## Material-file

The material file should contain a single JSON object, where each field of the object refers to a specific material definition. A material definition is a JSON object with two fields, **specular** and **diffuse**, both of which are arrays of eight floating-point values. The values in each array refer to coefficients in each of eight frequency bands, from low to high, and are used to calculate ray attenuation in each of these bands. Each material in the 3D model should have a corresponding field with the same name as the material in the material file.

An example material file is shown below:

```

{
  "concrete_floor":
  {
    "specular": [0.99, 0.97, 0.95, 0.98, 0.98, 0.98, 0.98, 0.98]
    , "diffuse": [0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6]
  }
  , "brickwork":
  {
    "specular": [0.99, 0.98, 0.98, 0.97, 0.97, 0.96, 0.96, 0.96]
    , "diffuse": [0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6]
  }
}

```