Implicit surfaces

Eduardo Rinaldi 1797800

1 Introduction

Until now we have always used **triangle or quad meshes** in our renderers; this extra credit aims to implement a renderer similar to the one implemented in the second homework but which can render scenes containing **ONLY implicit surfaces**.
In particular, this implementation allows to render scenes composed by **SDF** (signed distance function), such as:

- In-app defined SDF (lambda functions).
- 3D Grid SDF (volume grid).

(examples and brief explanation of both later)

1.1 Functionalities implemented

- Implicit surfaces shader.
- Very simple multiple importance sampling.
- Implicit normals shader for debugging.
- Spheretracing functions.
- Normal evaluation functions.
- `".sdf"` file parsing.
- Implicit scene loading using `"scene.json"`.

2 Code organization

The code tries to follow as closely as possible the style already defined in Yocto/GL, below there is a section describing how the code has been organised for each functionality.

2.1 SDFunction vs SDField grid

When we talk about implicit surfaces we mean surfaces defined by a function that measures the distance between a point and the surface.

$$S = \{p | sdf(p) = 0\}$$

In practice, these functions can be defined in two ways in our program:

1. As actual functions, i.e. user defined functions (easy to write), I'll call these *signed distance function* or *sdf*.
2. As 3D grids containing in each voxel the distance between the point and the surface, as if they were textures to be sampled (trickier to write), I'll call these *signed distance field grid* or *sdf grid*.

For allowing an easier scene creation experience I decided to implement in `yocto_sceneio.cpp` a modified version of the json parser that will allow us to define sdf (in both way, more or less..) directly in the scene file.

2.1.1 Signed distance function

I defined these in `yocto_scene.h:194` as:

```
struct sdf_data
{
    std::function<float(const vec3f&)> f;
    vec3f whd; // width-height-depth
    frame3f frame;
    int material = invalidid;
};
```

Where:

- `f`: is the actual sdf, many of these can be found here, I implemented a few of these in `yocto_sdfs.h`.
- `whd`: stands for width, height and depth, it's a parameter used for area calculation in MIS.
- `frame`: frame associated to the function, this allows us to rotate and translate sdf as we want.
- `material`: id of the material associated.

In the scene file we can define them using the following structure:

```
"sdfunctions": [
    {
        "name": "...", // name of the sdf
        "type": "box|bbox|capped_cone|plane|sphere|torus", // sdf type
        "material": n, // material id
        "frame": [...], // sdf frame
        // extra parameters here specific to each sdf, example sphere:
        "radius" 0.1
    },
    //others
]
```

Extra sdf can be definied at `yocto_sceneio.cpp:3685`; there you can also look at "extra parameters" for each sdf.

2.1.2 Signed distance field grid

I defined these in `yocto_scene.h:202` as:

```
template<typename T>
struct volume {
    vec3i whd;
    std::vector<T> vol;
    float res;
};

struct volume_instance {
  int volume = invalidid;
  int material = invalidid;
  float scalef = 1;
  frame3f frame;
};
```

Where:

- `struct volume<T>`: is just like a 3d texture; I looked into old versions of Yocto and there was a similar struct with some functions already defined, so I decided to create a similar struct for reusing part of these functions
  - `whd`: width, height, depth of the 3d grid.
  - `vol`: 1D list containing all the values of the grid.
  - `res`: resolution of the grid, is a parameter that is used for scaling the value contained in each voxel.
- `struct volume_instance`: the idea is similar to the one behind `yocto::instance_data`
  - `volume`: id of the volume.
  - `material`: id of the material.

- `scalef`: scale factor used for scaling the instanced volume.
- `frame`: frame of the instance, this allows us to rotate and translate the instance.

In the scene file we can load this from `".sdf"` files, using the following structure:

```
"volumes": [
    {
      "name": "...",
      "uri": "path/to/file.sdf", // relative path
      "binary": false // true if file is in binary format
    },
   // others...
  ],
  "vol_instances": [
    {
      "name": "...",
      "volume": 0, // Volume id
      "material": 3, // Material id
      "scale": 0.001, // Scale factor
      "frame": [...] // frame
    },
    // others...
]
```

The sdf parser can read only files generated by these two **sdf generators**:

1. `skanti/generate-watertight-meshes-and-sdf-grids` `[GitHub repo]`: to load files generated from this generator set `binary: true`
2. `christopherbatty/SDFGen` `[GitHub repo]`: to load files generated from this generator set `binary: false`

(parser can be found at `yocto_sceneio.cpp:885` as `load_volume(...)`)

How to retrieve the distance value for a given point p Given a point p we do the following things, in order:

- Obtain distance d_{bbox} of p from the bounding box of the grid (bounding box is easily calculated using `whd` and `res`).
- If $d_{bbox} > \epsilon$ we didn't hit the bounding box, so we can return d_{bbox}
- Othewise we hit bounding box, and now we can evaluate the volume using uvw coordinates (these are calculated by transforming $p \rightarrow p_{local}$, using instance frame).

(Code about this can be found at `yocto_sdfs.cpp:30` as `eval_sdf(volume, instance, p, t)`)

2.2 SDF scene evaluation

The scene evaluation at a given point p is done by doing a "minimum search" across all the SDFs (both functions and grids); more specifically we look for the minimum distance from p to each surface.
Is not the best way to proceed, indeed there are some accelerating structure that can be used for speeding up the code; for this extra credit I decided to focus on the grid evaluation and on creating a more general system that could fit well in the library as it is right now (i.e. scene parser, both sdfunction and sdfield grid)

2.3 SDF normals

For calculating the normal of an SDF at a given point p we have to take the gradient of it at point p.

$$n = normalize(\nabla f(p))$$

For doing this I'll use a numerical method explained here.
(Normal evaluation code can be found in `yocto_sdfs.cpp:51`)

2.4 Spheretracing

Spheretracing algorithm is very similar to the one proposed on professor slides. The only differences stands in the termination condition; indeed I decided to use a for loop limiting the maximum iterations to a fixed number.
This decision comes from the fact that when we have rays almost parallel to a surface the algorithm will slow down by a lot. The idea of this solution is taken from Inigo Quilez shader toy examples.

At the end of a call of spheretracing we will return the following information:

- `hit`: boolean that tells is we hit something
- `dist`: ray distance
- `instance`: id of the volume "hit"
- `sdf`: id of the sdfunction "hit"

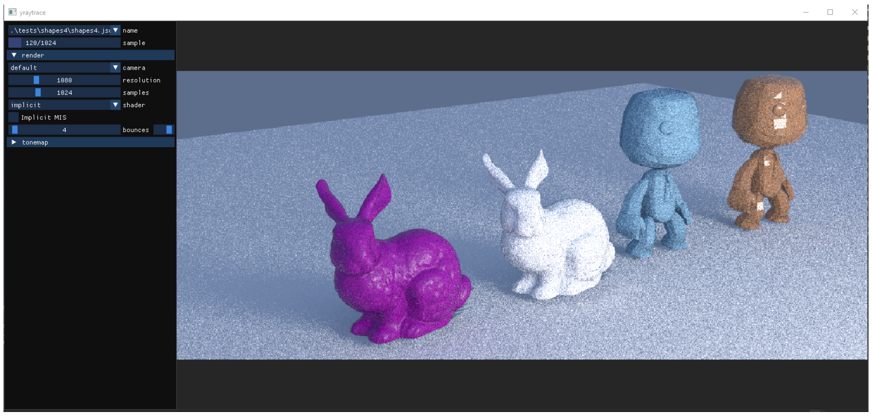
(Code about spheretracing can be found in `yocto_pathtrace.cpp:266`)

2.5 MIS

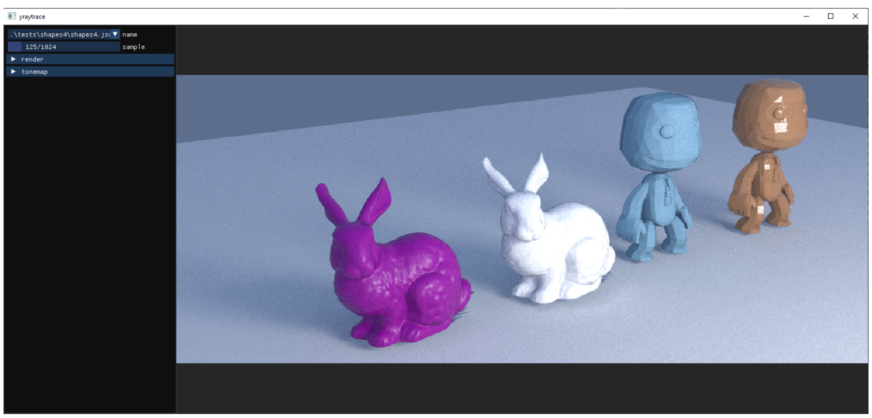
A complete MIS implementation is not that easy on implicit surfaces, but just for speeding up the rendering convergence time on some scenes I decided to implement a simple version of it.
This simpler version consists in only sampling quad lights defined as sdfunction, so that the sample function and pdf calculation function are easy to implement;

the idea is to generate a random uv, scale it to quad size and then use the frame for converting the generated point in world coordinate.
(Implementation of these two functions can be found in `yocto_pathtrace.cpp:312` in `sample_lights(...)` and `sample_lights_pdf(...)`)

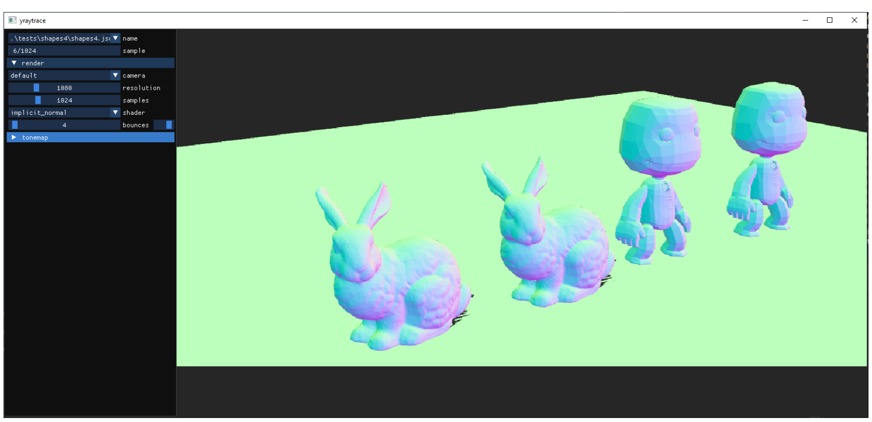
3 Results



(a) without MIS



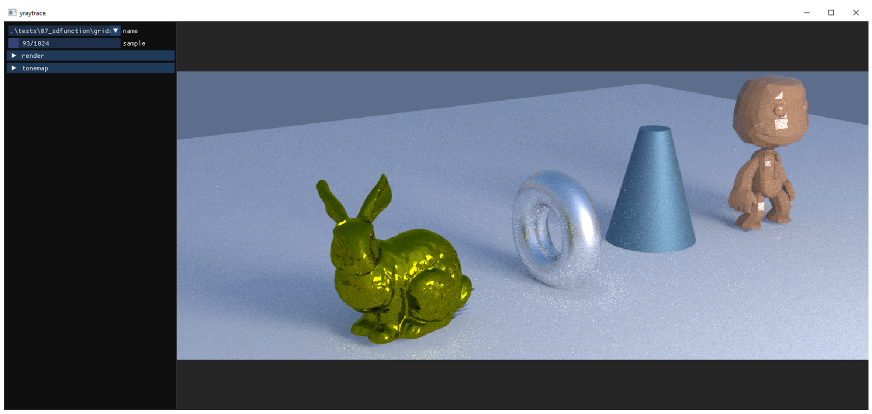
(b) with MIS



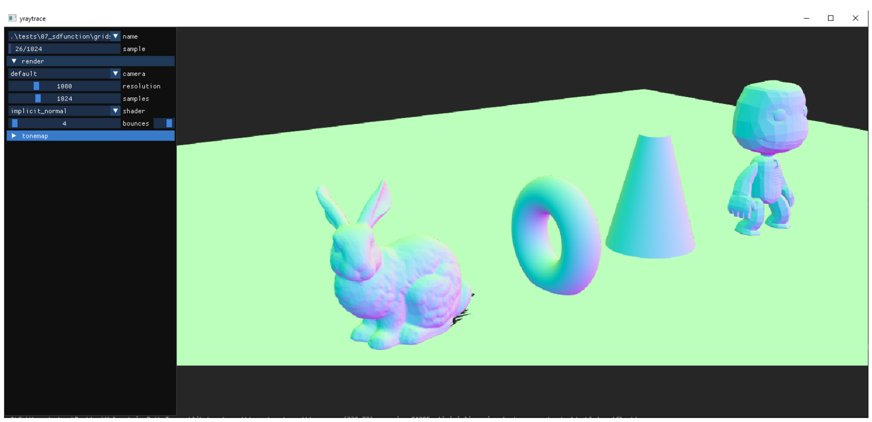
(c) normals

Figure 1: SDField grids only

As we can see the bunny is not well ”intersected”, it seems to be a problem with that specific SDF grid, since with others I don’t have this problem and generating another bunny SDF grid with differnet resolution almost fix the problem.



(a) with MIS



(b) normals

Figure 2: SDField grids and SDFunctions

These results are stored in `"out/extra/"` directory

4 How to test the code

For testing the code I have created two sample scenes that can be found in `"./tests"` as `"06_gridsdf"` and `"07_sdfunction"`.
Now we have two options:

- **CLI**: `ypathtrace --scene scene --output output --shader ("implicit"|"implicit_normal")`

- **Interactive** (suggested): `ypathtrace --scene scene --output output --shader ("implicit"|"implicit_normal") --interactive`