真实感渲染技术大作业报告

孙子平 2020213960

2021 年 6 月 27 日

1 项目简介

本项目采用光线追踪的方式进行真实 3D 渲染。渲染的核心部分采用 Rust 编写，部分 API 可被 Python 调用，借此我实现了两种使用本项目的方式：

- 在命令行里通过 JSON 导入配置，无图形界面地渲染；
- 在 PyQt5 的界面里配置，并有实时预览（由于时间受限，许多 API 还没导出给 Python 使用）。

本项目最大的特点在于性能，通过同时借助线程池和 SIMD 技术，使得在不借助显卡的情况下，将 CPU 的性能发挥到极致。

此外本项目还有很多的形状、材料、纹理。其中形状包括：

- 球体；
- 与坐标轴平齐的矩形；
- 按固定概率散射光线的雾；
- 支持法向和纹理坐标的三角面；
- 与坐标轴平齐的立方体（矩形的组合）；
- 网格（三角面的组合）。

每个形状都能进行任意的旋转，不均匀的缩放和位移。这使得球体能成为长轴任意方向的椭球；坐标平齐的矩形成为任意方向的矩形。本项目支持的材料包括：

- 支持贴图的朗伯反射体（漫反射）；
- 随机散射（用于雾）；
- 电介质（有折射和反射，如玻璃）；
- 面光源；
- 金属（有模糊程度的镜面反射）

本项目支持的纹理包括：

- 常值颜色；
- 来回切换两种纹理；

- 图像贴图;
- Perlin 噪音

本项目只实现了透视相机，相机支持光圈、焦距等设置，可以自由地选择位置。渲染支持设置背景色、抗锯齿。

2 如何构建及使用本项目

2.1 构建项目

项目核心是 Rust 编写的。在编译前，需要确认已经安装 rust 构建系统。推荐使用 rustup 这个工具管理 rust 版本。我们主要要编译两个东西，一个是命令行的渲染工具，另一个是给 Python 调用的动态链接库。

给 Python 调用的动态链接库

```
# 首先切换到项目的根目录
cargo build --lib --release --features python
```

如果为了追求性能，对于某些已经有 AVX2 指令集的 CPU，可以编译成最快速且适用于当前主机 CPU 的指令。方法是添加环境变量 `RUSTFLAGS=-C target-cpu=native`。某些平台可能出现链接错误，需要手动添加 Python 动态链接库，通常是不需要的。

由于各个系统的 Python 版本不一样，所以无法给出一个很适合的二进制版本。

编译命令行渲染工具

```
# 首先切换到项目的根目录
cargo build --bin ray-tracing --release
```

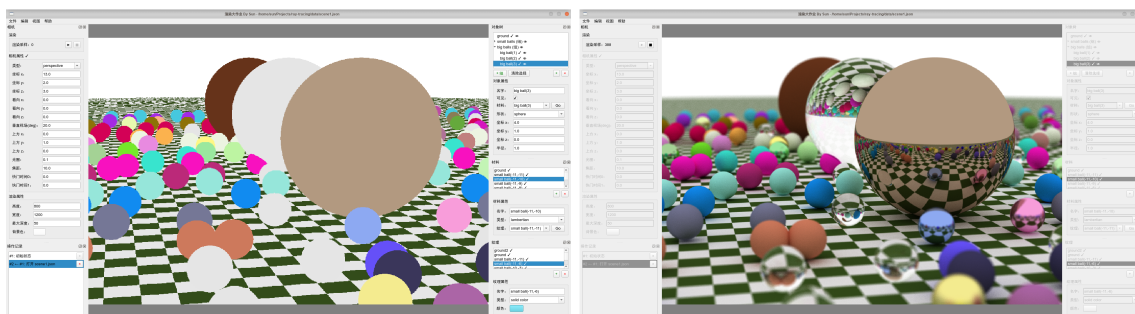
执行上面的命令就可以编译出命令行的渲染工具了。我在`./bin` 目录下给出了预先在 Windows 上和 Ubuntu 18.04 上编译的版本。

2.2 运行项目

Python 前端 编译完命令行渲染工具后,在 Windows 上需要将`./target/release/v4ray.dll` 拷贝到项目根目录,并重命名为 `v4ray.pyd`;在 Linux 上需要将`./target/release/libv4ray.so` 拷贝到项目根目录，并重命名为 `v4ray.so`。

```
# 安装依赖
pip install PyQt5 numpy
python main.py
```

之后你就可以看到前端了，效果详见图 1。在这个前端里，你可以添加纹理、材料和对象。对象可以分组，以方便批量开关显示以及批量指定材料。由于 deadline 前一两天，我大规模地



(a) 打开文件后的界面　　(b) 正在渲染中的界面

图 1: Python 前端界面

改变了 API。所以 Python 的 API 是使用旧的，出于展示保留的 API。功能也很受限，所以形状只有球，材料只有金属、电介质、朗博反射体，纹理只有单一色和交替切换。

在还没有渲染之前，主界面会显示预览。渲染之后，主界面显示渲染结果。此外这个前端还能打开、保存项目，导出渲染的结果。最后，这个前端支持无限的撤销重做。

编译命令行渲染工具　以下是这个项目的帮助。命令行工具会以--input 的参数作为输入的文件，输出到--output 的图片文件中。在运行时，当你敲入换行后，程序就会干净地退出。程序每隔 5 秒会保存一次结果到文件中。每成功采样一次，程序都会打印当前迭代次数和总共耗时，命令行输出类似图 2。

```
ray-tracing 0.1.0
Ziping Sun <me@szp.io>
as do all doc strings on fields

USAGE:
    ray-tracing-linux-x86_64 [OPTIONS]

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -i, --input <input>      [default: scene.json]
    -o, --output <output>    [default: output.bmp]
```

在./bin 目录下，我们已经内置了 5 个渲染后的场景。场景渲染的结果按照 scene<n>-<iter>-<sec>.bmp 进行命名。之后我会详细介绍。关于 JSON 文件的格式，我相信例子比定义更方便，./bin/scene0.json 包含了所有的形状、材质等等，是个很好的参照。需要指出的是 JSON 的根可以包含可选的，带 name 的 shapes、materials 和 textures。通过直接在需要形状、材质出现的地方直接填写对应的 name，可以拆解对象，重复利用一些设置。

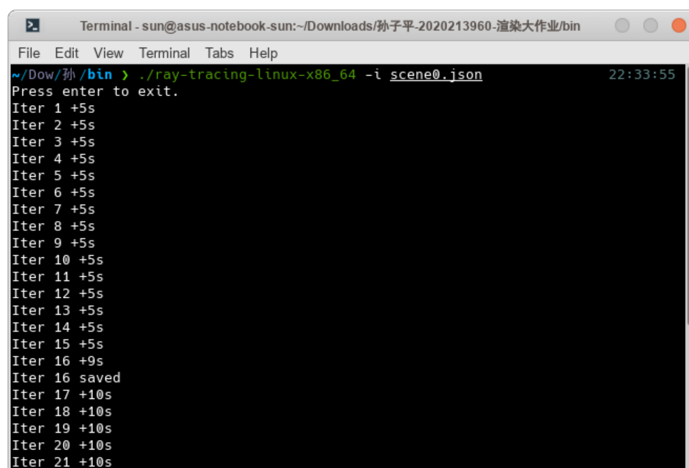


图 2: 命令行界面

3 渲染结果展示

3.1 场景 1

场景 1 的效果见图 3。这个场景就是杂七杂八的东西都塞进图里做出来的。主要展示了球体、长方体、网格、贴图、玻璃、面光源、重要性采样等等工作得正确与否。

3.2 场景 2

场景 2 的效果见图 4。这个图展示了朗博反射体、电介质、金属的表现。这张图很好看，就放进来了。

3.3 场景 3

场景 3 的效果见图 5。这个主要展示了金属感的 Bunny 的高光。

3.4 场景 4

场景 4 的效果见图 6。这个主要是演示载入两个模型的功能。

3.5 场景 5

场景 5 的效果见图 7。又一个 Cornell Box。这个画面和某一个同学以前的 C++ 光追实现很像。我主要用着个来看看我的 SIMD 版能快多少。实测大概快 3 倍左右。

4 项目特色：线程池和 SIMD

线程池技术我还是用得比较简单的。为了避免频繁地访问锁，我就每个线程负责采一次样。这样做的缺点是，第一帧会很晚才来，而且如果渲染比较耗时，在程序刚开始时，线程就几乎

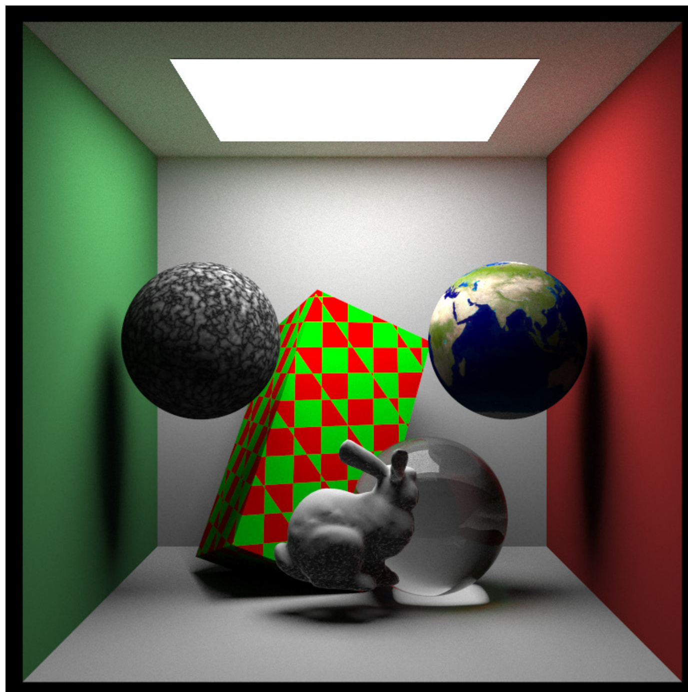


图 3: 场景 1

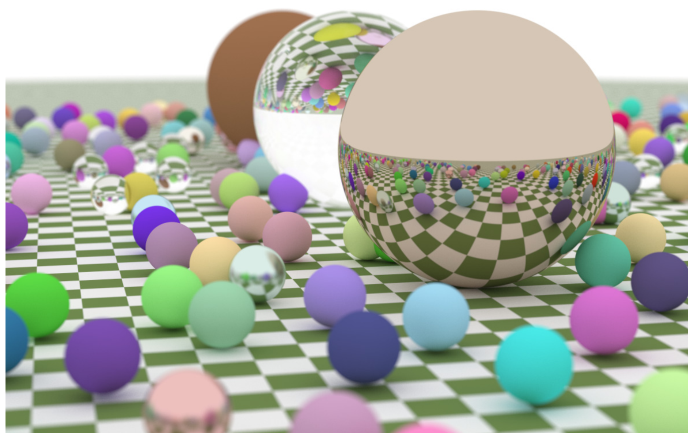


图 4: 场景 2

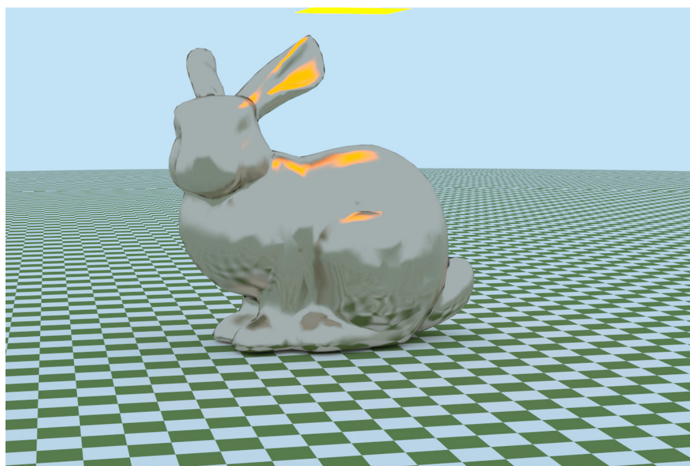


图 5: 场景 3

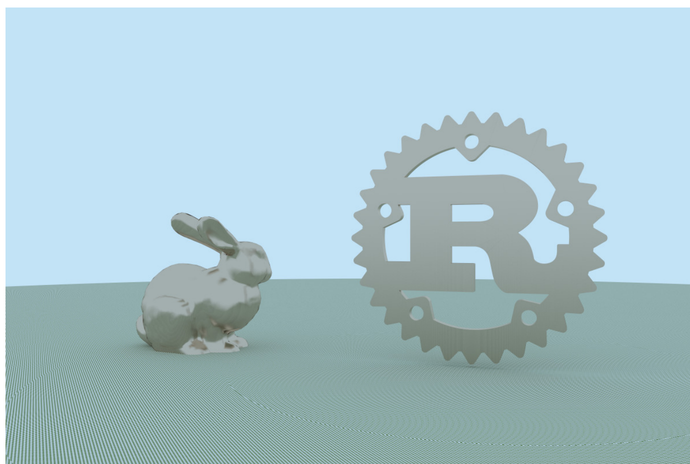


图 6: 场景 4

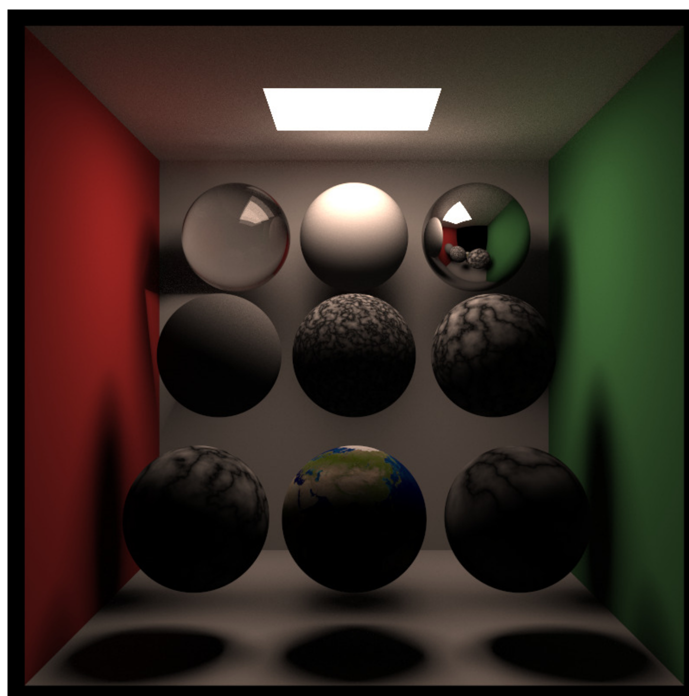


图 7: 场景 5

同时完成。

SIMD 技术就比较麻烦了。这里我用的是 AoSoA 的数据结构。借助泛型，我的代码只需要一处修改，就能在 AVX、AVX2、AVX512 等指令集切换。默认我采用 AVX2。在实现过程中，遇到最困难的是光线在完成 BVH 粗略的碰撞检测后，需要重新排序，使得同一碰撞物体的能批量精细碰撞检测；之后还需要再排序，按照最近碰撞的进行散射等等。总之，代码经常需要对光线排序，以最大化并发，这部分代码写的比较麻烦。

4.1 最后

如果时间更多，我可以实现平行透视，加入更多的元素，并且写出个人性化的界面。但无论如哦，我已经学习到了很多知识，即使时间那么短。