LearnOptix 系列 4-光线追踪器的优化与交互

Dezeming Family

2023 年 4 月 24 日

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列书，可以从我们的网站 [https://dezeming.top/] 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

2023-4-25：完成第一版。

目录

一　本文介绍

本文的内容相比于上一篇简单不少，只是做点儿优化。我们针对三点内容来做点儿优化：

- 第一是实现一个累加缓冲区，将多个帧的结果进行合并，得到更好的输出结果；
- 第二是实现一个可以移动的 FPS 视角的相机；
- 第三是实现棋盘地板；

我们分为三个章节来描述。

二　缓冲区

本节代码见 2-1 目录。

我们的缓冲区要稍微高级一点了，实现历史帧保存和结果累加功能。

我们需要创建一个存储历史帧的 Buffer。以及记录一个帧变量（记录当前是第几帧），当图像缩放时，帧变量清零；后面当移动相机时，帧变量也需要清零。

```
context["frame"]->setUint( 0u );
Buffer accum_buffer = context->createBuffer( RT_BUFFER_INPUT_OUTPUT |
    RT_BUFFER_GPU_LOCAL,
RT_FORMAT_FLOAT4, width, height );
context["accum_buffer"]->set( accum_buffer );
```

其中，两个标志位的意义是：

```
// Buffer既能写入到GPU，也能从GPU读出
RT_BUFFER_INPUT_OUTPUT
// 该标志只能跟RT_BUFFER_INPUT_OUTPUT配合使用，限制主机端写入到GPU中。当有多
    个设备时，读写访问都是在每个设备上运行的副本。
RT_BUFFER_GPU_LOCAL
```

glutResize() 函数需要注意 resize 这个 Buffer：

```
sutil::resizeBuffer( context[ "accum_buffer" ]->getBuffer(), width, height )
    ;
```

定义一个静态变量 accumulation_frame 用来记录当前帧是第几帧，当窗口 resize 时，该值归 0。在每次循环调用 glutDisplay() 函数时，该值都被传递给 GPU 中的 frame 变量：

```
static unsigned int accumulation_frame = 0;
if (is_WH_changed) {
    updateCamera();
    accumulation_frame = 0;
    is_WH_changed = false;
}
context["frame"]->setUint(accumulation_frame++);
```

在 camera.cu 文件里需要先声明：

```
rtBuffer<float4, 2>              accum_buffer;
rtDeclareVariable(unsigned int,  frame, , );
```

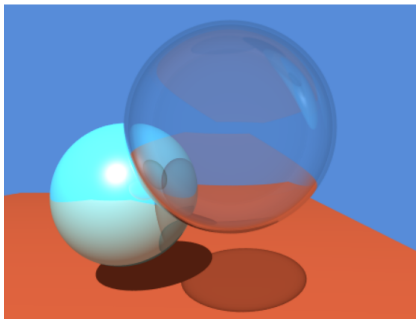
camera.cu 文件的函数里需要增加点内容，首先是根据屏幕坐标和当前帧生成一个种子点，然后根据种子值来调用 rng 函数生成 rng 随机数，此随机数用来做子像素抖动：

```
size_t2 screen = result_buffer.size();
unsigned int seed = tea<16>(screen.x*launch_index.y+launch_index.x, frame);
float2 subpixel_jitter = frame == 0 ? make_float2(0.0f, 0.0f) : make_float2(
    rnd( seed ) - 0.5f, rnd( seed ) - 0.5f);
```

其他内容不变，最后将生成的结果保存到累积 Buffer 内：

```
float4 acc_val = accum_buffer[launch_index];
if( frame > 0 ) {
    acc_val = lerp( acc_val, make_float4( prd.result, 0.f), 1.0f /
        static_cast<float>( frame+1 ) );
} else {
    acc_val = make_float4(prd.result, 0.f);
}
output_buffer[launch_index] = make_color( make_float3( acc_val ) );
accum_buffer[launch_index] = acc_val;
```

渲染多帧以后的结果的边缘就没有锯齿了：



三　相机交互

本节代码见 3-1 目录。

我们在 main.cpp 中设置一个变量 camera_dirty，这是一个 bool 类型的变量，用来记录画布是不是需要重置（当相机方向改变，或者画布尺寸改变，该值都要置位为 true，然后使 frame 值归 0）。

定义两个鼠标状态位，记录鼠标是否按下，以及鼠标之前抬起时的位置：

```
1 // Mouse state
2 int         mouse_button;
3 int2        mouse_prev_pos;
```

然后实现下面两个回调函数，用户鼠标点击和移动时的响应：

```
1 void glutMousePress( int button, int state, int x, int y );
2 void glutMouseMotion( int x, int y);
```

在 glutRun() 函数中注册（注意，glutMotionFunc 注册的函数只有在鼠标按下时才会响应）：

```
1 glutMouseFunc( glutMousePress );
2 glutMotionFunc( glutMouseMotion );
```

鼠标移动旋转相机的实现就在 glutMouseMotion() 函数里，我们是根据 LearningOpenGL[10] 实现的，camera_front 表示相机的前方。在我们的实现中，按下鼠标左键移动鼠标，可以自由移动视角；按下鼠标右键移动鼠标，可以前进和后退。

四　棋盘地板

本节代码见 4-1 目录。

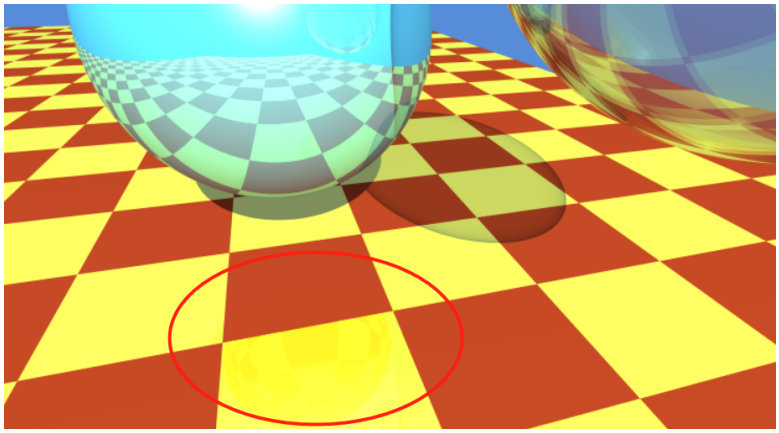
parallelogram.cu 中会计算出纹理坐标：

```
rtDeclareVariable(float3, texcoord, attribute texcoord, );
// intersect()函数
texcoord = make_float3(a1,a2,0);
```

其中，a1 和 a2 都是 0 到 1 之间的数字。

棋盘格实现在 checker.cu 文件里，其实就是两种 Phong 材质，根据纹理坐标计算出属于哪种材质，然后计算着色即可。在 main.cpp 的 createGeometry() 中也要注意设置一下参数。

渲染结果如下：



可以注意到，黄格子的 Phong 材质 Kr 大于 0，所以会有反射。

五　小结

本文的实现较为简单，关键在于更好地理解和熟悉 Optix 编程方法。

下一篇文章中，我们将会介绍 Optix 中比较难的部分——纹理。纹理不但包含坐标采样，还涉及多级纹理、纹理滤波等比较高级的技术。掌握纹理以后，我们渲染的图像就可以更好看了。

参考文献

[1] https://developer.nvidia.com/rtx/ray-tracing

[2] https://developer.nvidia.com/rtx/ray-tracing/optix

[3] https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/

[4] https://raytracing-docs.nvidia.com/optix7/index.html

[5] https://raytracing-docs.nvidia.com/optix7/guide/index.html#preface#

[6] https://developer.nvidia.com/designworks/optix/downloads/legacy

[7] https://raytracing-docs.nvidia.com/optix6/guide_6_5/index.html#guide#

[8] https://raytracing-docs.nvidia.com/optix6/api_6_5/index.html

[9] https://raytracing.github.io/books/RayTracingInOneWeekend.html

[10] https://learnopengl.com/Getting-started/Camera