山东大学 计算机科学与技术 学院

计算机图形学 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200130119 | 姓名：于斐 | | 班级：学堂计机22 |
| 实验题目：Ray tracing | | | |
| 实验学时： 4 | | 实验日期： 2024年11月23日 | |
| 实验目的：  通过实现光线追踪算法，在OpenGL环境下构建一个真实感3D场景，深入理解光线追踪的核心原理和渲染流程。实验旨在掌握光线与几何体（如球体、平面）交互的计算方法，以及通过光源与相机位置的调节，生成逼真的光影效果。此外，通过实现场景中的颜色、反射和阴影等细节，体会基于物理的光照模型的魅力，为后续在计算机图形学领域的深入学习打下基础。 | | | |
| 实验步骤与内容：  实验环境：OpenGL 4.6 及 GLFW, GLM 等附属库。  实验步骤：   1. 与实验1完全相同：项目使用 CMake 管理，原则上任何支持CMake的编辑器均可使用。所有未包含在项目文件中的库均使用 CMake 的FetchContent导入，不需额外手动安装任何库。 程序自动生成到 dist 目录下，但运行时pwd需包含assets。因此需要在根目录运行 dist/Renderer 或将 assets 拷贝到 dist 下。    1. 建立OpenGL窗体及 argument parser, config parser。前者代码位于 Renderer::init()中，在 src/renderer/renderer.h, renderer.cpp 下可以找到。后者代码位于 src/utils 中。使用 ImGUI 构建交互选单，用于控制程序运行时的行为。相关代码位于 src/gui/ 下。 2. 普通光线追踪方法分析 考虑一个普通光线追踪方法。此处一步到位考虑 path tracing。     图片引用自 <https://en.wikipedia.org/wiki/Path_tracing>  上面的方法在 CPU 上很好实现。但是在 GPU 上有困难。首先，OpenGL 的传统的 vert + frag shader 似乎不适合完成这个任务。其次，过程中存在递归，而递归在 GPU 中是不兼容的。  普通的光线追踪运算瓶颈在求交上。一般可使用 BVH 等数据结构加速。但是，BVH 在 traverse的过程中也可能存在递归，同样需要处理。   1. 任务分析   然而，本次实验任务与一般的光线追踪任务不同。本次实验任务的场景很简单。只有两个球和一个房间，后者最多 12 个三角形面。这个场景包含的primitive 数量远小于普通场景，因此完全不必构建 BVH 等求交加速结构（BVH 也完全没有办法达到加速效果），仅需暴力判断即可。  考虑本次任务的运算瓶颈，目前的瓶颈就仅存在于 foreach pixel 上了。最好的办法是使用一个通用的并行运算结构，输入数据以后每个 pixel 分配给一个线程，单独求交。  OpenGL 自 4.3 起支持 Compute Shader。这是一个通用运算 shader，支持输入 buffer 后按照类似 cuda 等并行运算结构的方式启动 kernel。这一项功能完美符合任务的要求，因此考虑使用 Compute Shader 作为主力运算单元。   1. Compute Shader   （实现位于 assets/shader/trace.comp）  首先讨论输入和输出。  对于输入，OpenGL 的 Compute Shader 支持 Buffer 形式的数据转移，因此我们将物体数据（墙和球）全部送入 glBuffer，即可被 Compute Shader 调用。  此处的一个难点是，OpenGL 的 struct 内存对齐规则与 CPU 规则不一致，因此需要严格处理 padding 问题。因此有这样滑稽的 struct 定义：    除此之外，相机位姿、光照等信息可使用 uniform 送入。  对于输出，使用 Image2D 输出到 OpenGL Texture即可。  对于并行部分，此并行任务相对简单，无需考虑 shared memory 等问题。对于每个 thread，直接使用 gl\_GlobalInvocationID 得到屏幕坐标即可。  到目前，要考虑的仅剩下实现部分。在实现层面，path tracing 除递归外的方法实现与 CPU 一致，包括球、三角形求交、shadow ray resolving等，此处不再赘述。  一个需要考虑的问题是随机数问题。OpenGL 不原生支持随机数，因此我们需要在每次 launch kernel 时传入一个种子，来做手动伪随机。我们使用了当前已经渲染的帧数来做种子，恰好这个变量需要在 denoising 部分使用。  考虑递归部分。不难注意到，如果我们每次反射时只产生一条光线，那么我们可以直接用循环代替递归，每次更新新的光线即可。类似许多光线追踪方法，我们使用俄罗斯轮盘赌等方法来做简单的加速，并在其中控制 radiance 保证蒙特卡洛估计无偏。      但是只有一条光线的 trace 结果显然是很 noisy 的。为了 denoise，与许多光线追踪渲染器类似，我们使用了 average over multiple frames 和 average over samples per pixel 的方法。    对于每一帧的每一个 pixel，我们采样 spp 次，每次在初始的 ray 上随机给一个 jitter。最终这一帧的结果是多次采样的 average。顺便，我们使用这个方法达到了 anti aliasing 的效果。  在场景固定后，我们 average over multiple frames，每次以一个越来越小的比例对 final color 取加权平均值，最终达到收敛。  我们最终的 Compute Shader 的主要伪代码如下：  function computeShaderMain()      // 与 cuda 等通用并行计算方式类似，得到当前线程要处理的坐标      pixelCoords = getGlobalInvocationID()      if pixelCoords out of bounds then return      // 从屏幕坐标转换到 uv 坐标      uv = normalizeScreenCoordinates(pixelCoords, width, height)        // 初始化随机数种子，使用了当前渲染的帧数      seed = initializeSeed(frame, pixelCoords)      // 多次采样，每次发射一个抖动光线，并做 path tracing      color = vec3(0.0)      for i = 0 to spp - 1 do          jitter = generateRandomOffset()          newRayDir = calculateRayDirection(perturbUV(uv, jitter), cameraFront, cameraRight, cameraUp, cameraFov)          color += tracePath(cameraPos, newRayDir)      color /= spp      // 与上一帧的结果混合，并逐渐收敛      lastRadiance = loadImage(outputImage, pixelCoords)      if rerender != 0 then lastRadiance = vec4(0.0)      color = (lastRadiance.rgb \* frame + color) / (frame + 1)      storeImage(outputImage, pixelCoords, vec4(color, 1.0))  end function  function tracePath(rayOrigin, rayDirection)      radiance = vec3(0.0)      throughput = vec3(1.0)      // 递归 path tracing。由于 OpenGL 不支持递归，此处使用循环代替，且光线不扩散，只有一次反射      for bounce = 0 to maxBounces - 1 do          // anyhit          hit, t, hitNormal, hitColor = intersectScene(rayOrigin, rayDirection)          if not hit then break          // closesthit          radiance += calculateDirectLighting(rayOrigin, rayDirection, hitNormal, hitColor)            // 更新光线方向，采样新的方向          rayDirection = sampleNewDirection(hitNormal)            // RR，随机终止路径          if terminatePathRandomly() then break          // 此处给 throughput 补偿，使得估计无偏          throughput \*= updateThroughput(hitColor, rayDirection)          // 更新光线起点          rayOrigin += rayDirection \* epsilon      return radiance  end function   1. 其他实现   在 CPU 层面，构建 Scene 类用于物体添加和光源管理。向其中添加五面墙（空一面用于进光）和两个球体。相关代码可在 src/ray\_tracer/scene.cpp 中查看。  针对 radiance resolving 的过程，实验没有要求严格的构建反射、折射材质，因此使用 color 近似 diffuse material，使用球型光源（因此包含 light radius 变量）。没有保证严格的数值与真实物理量的一一对应关系。  在 GPU 层面，最终使用一个很简单的一一对应 shader 将 path tracing 结果写入屏幕。过程中使用 tone mapping 做颜色映射。OpenGL 自动处理 gamma correction。  整个 trace 过程可以达到很高的效率，在1080P，Max. Bounces = 5, SPP = 4 的情况下，在 RTX3050 上运行，可以达到 120fps。此时 GPU 占用约为 50%，猜测有一大部分开销来自于 Device – Host 的数据交换。在 SPP = 10 的情况下，GPU 占用可接近 100%，此时可以达到 ~30fps。但此时 SPP 已经完全冗余。   1. Demo | | | |
| 实验总结：  本次实验成功实现了一个光线追踪渲染器，并构建了包含两个球体和房间的3D场景。通过使用光线与球体、平面的交点计算，以及基于法线的光照计算，生成了真实感的阴影和光影效果。实验中，通过调整光源位置和观察点，观察了场景的不同渲染结果，进一步理解了光线追踪的灵活性和真实感表现力。本次实验学习了 OpenGL Compute Shader 的使用方法，对 GPU 通用计算有了更深刻的理解。 | | | |