计算机动画的算法与技术 碰撞检测大作业 结题文档

软件 03 陈启乾 2020012385

1. 说明程序运行环境，以及项目和代码依赖的编程环境

1.1. 本机程序运行环境

运行环境：

- Windows 11
- Vulkan 图形后端

理论可以在所有操作系统，以及任何支持 Vulkan/Metal/OpenGL/DX 的图形后端上运行。

1.2. 编程环境及配置方法

1. Rust 1.74.1：可以从 Rust 官网下载 https://www.rust-lang.org/zh-CN/learn/get-started
2. 其他依赖包以 `cargo.toml` 中为准，以 `wgpu` 作为图形后端，`winit` 作为窗口后端。
3. 在项目根目录下运行 `cargo run --release` 即可编译运行，二进制在 `target/release` 目录下

（因为编译后的二进制依赖本地资源，所以无法直接运行编译后的二进制文件）

2. 各个程序模块之间的逻辑关系

各个模块均位于 `src` 目录下，具体介绍如下：

2.1. Camera 模块

维护 `position`, `yaw` 和 `pitch`。

在用户输入（拖拽界面/滚轮）的时候，更新 `yaw` 和 `pitch`；在用户输入（按键）的时候，更新 `position`。

在渲染前将 `position`, `yaw` 和 `pitch` 转换为 `view` 矩阵，作为 Uniform Buffer 传入渲染管线。

2.2. Light 模块

维护 `position` 和 `color`，分别表示光源的位置和颜色。

在渲染前将 `position` 和 `color` 转换为 `light` 矩阵，作为 Uniform Buffer 传入渲染管线。

2.3. Framework 模块

主要是与 Event Loop 相关的代码，创建窗口以及后端实例，处理窗口事件。

当 `Event::RedrawRequested` 事件发生时，会调用 `State::update` 更新状态，再调用 `State::render` 渲染。

当 `Event::WindowEvent` 事件发生时，根据具体事件，调用 Camera 和 Light 的方法，更新摄像机和光源的状态。

2.4. Resources 模块

从文件加载模型、纹理到 Rust 对象的模块。

2.5. Texture 模块

维护纹理相关功能，包括：

- 创建 Depth Texture
- 从 `image::DynamicImage` 创建 Texture, Sampler, ImageView，并且写入 Texture Buffer

2.6. Model 模块

维护模型相关功能，包括：

- 将 `obj` 文件解析为 `Model` 对象
- 将 `obj` 文件内的 Mesh 转化为 `Mesh` 对象，将 `mtl` 文件内的 Material 转化为 `Material` 对象
- 将 `Mesh` 对象转换为 Vertex Buffer 和 Index Buffer
- 维护 `draw` 方法，绘制 Model（包括实例化绘制）

同时还维护了光源相关的绘制方法，包括 `draw_light_model` 和 `draw_light_model_instance`。

2.7. Instance 模块

这里主要维护实例化绘制的 Buffer 以及信息，会在每次绘制所有小球前将 Instance 的 Vertex Buffer 更新；以及接收从主模块传入的小球信息，将其转换为 Instance 的 Vertex Buffer。

2.8. Main 模块

这里主要维护主模块的状态，会在主事件循环中被更新以及维护，包括：

1. 渲染物体的管线：`render_pipeline`
2. 渲染光源的管线：`light_render_pipeline`
3. 模型：`obj_model`
4. 深度纹理：`depth_texture`
5. 摄像机的状态：`camera_state`
6. 光源的状态：`light_state`
7. 实例化绘制的状态：`instance_state`
8. 计算模块的状态：`compute_state`
9. 上一次更新 FPS 的时间：`last_fps_update`

在创建(`State::New`) 时候，我们分别递归创建以上各个状态，然后将其传入 `State` 对象中。

在更新(`State::Update`) 时候，我们会调用以上各个状态的更新函数，包括更新摄像机的状态，更新光源的状态，计算碰撞检测的结果，并且用碰撞检测的结果去更新实例化绘制的状态。

在绘制(`State::Render`) 时候，我们会调用以上各个状态的绘制函数，包括绘制物体，绘制光源，绘制实例化绘制的小球。

除此之外，还维护了与窗口变化相关的功能。

2.9. Compute 模块

这里主要维护碰撞检测和处理相关的功能。

我们通过 WGSL 中的 Compute Shader 功能来实现对 GPU 的编程

3. 程序运行的主要流程

程序运行的主要逻辑位于 `main.rs` 和 `compute.rs` 中。

3.1. 初始化：生成

所有的小球会被限定在一个 $[-\text{boundary}, \text{boundary}]$ 的三维空间中，给与一个初速度，随机生成在空间中。

我们默认重力为 $[0, 0, -9.8]$，空气阻力系数为 0.1。

3.2. 物理引擎（碰撞检测，碰撞处理，位置更新）

每次在某一帧更新前，我们都会进行物理引擎的更新。为了保证精确，我们会在每帧之间的时间内进行多次小时间间隔的物理状态的迭代，并且以最后一次迭代的结果作为这一帧的状态渲染。

3.2.1. 碰撞检测

朴素的碰撞检测算法中，我们会对所有物体两两之间进行碰撞检测，这样的话需要进行 $O(n^2)$ 次碰撞检测。在 n 很大的时候，这样的算法效率会比较低。我们的碰撞检测算法，是一个两阶段的碰撞检测算法：

1. 粗检测阶段：在这一阶段中，我们希望能够通过较为简单、易于维护的数据结构，尽可能减少需要进行检测的物体对的数量

   因为题目中要求的是大量小球，可以假设物体大小相差不会很悬殊，而且大多数物体都是比较规则的几何形状。

   因此，我们打算采用基于均匀网格技术的粗检测方法。假设所有的物体中，AABB 包围盒最大的大小为 d，则我们在整个空间构造宽度为 $2d$ 的网格，每个网格的大小为 d。这样的话，我们可以将所有的物体放入网格中，每个网格中的物体数量不会太多。

   因为物体的大小不会超过网格大小，所以一个物体最多只会和其所在的网格为中心的 $3 \times 3 \times 3$ 个网格中的物体发生碰撞，在这些之外的物体就不会在碰撞检测的考虑范围内。

   假设物体相对均匀分布，那么我们就只需要进行 $O(n)$ 次碰撞检测。

   在粗检测阶段，我们希望维护一个索引数组，在索引数组中，我们把所有物体的索引按照其所在的网格进行排序；除此之外，我们还希望维护每一个网格在索引数组中的起始位置和终止位置。

   在第一个 Compute Shader（`shaders/assign.wsgl`）中，我们会并行计算出每个小球 Instance 所在的格的编号。

   在第二个 Compute Shader（`shaders/sort.wsgl`）中，我们会对所有的小球 Instance，按照其所在的格的编号进行排序，这样的话，我们就可以保证所有的小球 Instance，按照其所在的格的编号，依次排列在索引数组中。我们使用双调排序（Bitonic Sort，一个并行的排序算法），可以在 $O(\log^2 n)$ 的时间内完成排序。

   在第三个 Compute Shader（`shaders/build_grid.wgsl`）中，我们会计算出每个网格在索引数组中的起始位置和终止位置。我们会并行地判断，排序后的每个小球 Instance 所在的 Cell 是否和其前一个小球 Instance 所在的 Cell 相同；如果不同，那么我们就找到了这个 Cell 在索引数组中的起始位置和终止位置，这个线程就会将这个信息写入到一个数组中。由于只有一个边界线程会写入信息，所以这个操作不会产生 Race Condition。

   这样我们就得到了每个 Cell 中的小球 Instance 信息，也就完成了对于空间的划分。

2. 细检测阶段：在这一阶段中，我们对粗检测阶段中判断对进行精确的碰撞检测，并用 GPU 进行并行化加速。

   在细检测阶段中，在第四个 Compute Shader 中(`shader/build_grid.wgsl`)，我们对粗检测阶段中判断对进行精确的碰撞检测，并用 GPU 对不同物体之间的碰撞，进行并行化加速。如果两个小球的距离小于两个小球的半径之和，那么我们就认为这两个小球发生了碰撞。

3.2.2. 碰撞处理

对于已经碰撞的物体，我们会在碰撞方向施加一定的冲量，让两者可以脱离。具体来说，我们会根据他们碰撞的深度，生成一个与碰撞深度成正比，沿着碰撞轴线方向相互离开的力。（可以类比弹力）。

对于每一个小球，我们会将所有与其碰撞的物体的冲量叠加，作为这个小球的冲量。

3.2.3. 位置更新

得到了冲量，我们可以计算出小球这一时刻的加速度。我们会用一阶近似来计算速度，用二阶近似来计算位置。以上两部分同样在第四个 Compute Shader 中计算完成。

在得到若干轮时间迭代后的最终结果后，我们会将结果 Buffer 映射（Map）到主机中，然后将新的时间和速度写回到小球的状态中。

3.3. 渲染

在计算阶段完成后，渲染的 Instance 模块会从 Compute 模块中获得所有小球的信息，然后根据位置计算出小球的变换矩阵，将其写入到 Instance Buffer 中。

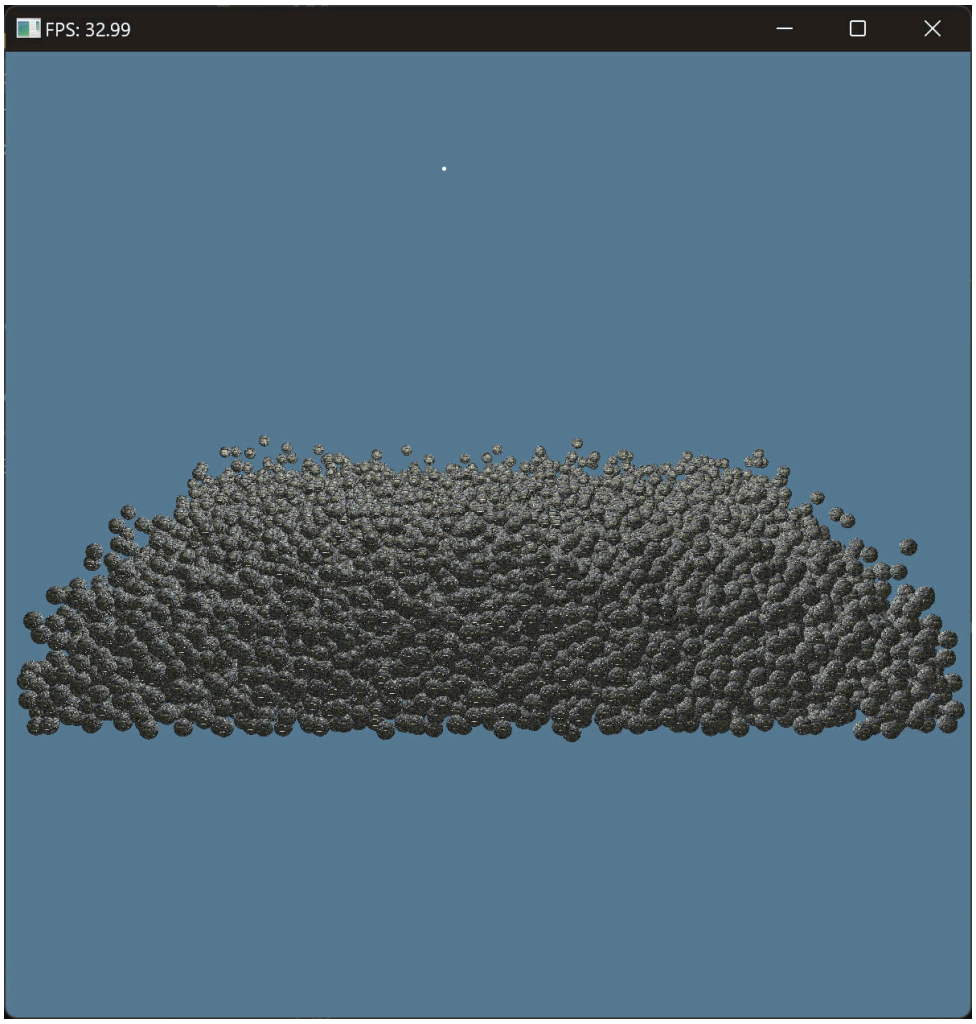
然后我们将 Instance Buffer 传入渲染管线，使用实例化渲染将所有小球渲染在屏幕上。

4. 简要说明各个功能的演示方法

运行程序后，即可看到许多球体在空间中运动，同时有一个光源在球体上方运动。

1. 使用 WASD 控制摄像机的移动
2. 使用鼠标在画面中拖动控制摄像机的视角。
3. 使用鼠标滚轮可以调整画面的缩放。

在窗口左上角会显示当前的渲染帧率，以 Frames Per Second(FPS) 为单位。



演示视频在 demo.mp4。由于 GPU 占用高，录屏清晰度较差，实际清晰度很高。

5000 个球可以达到约 40 FPS。

5. 参考文献或引用代码出处

1. Learn WGPU: https://github.com/jinleili/learn-wgpu-zh/ (参考了基本的 WGPU 框架语法以及窗口和渲染的基本框架)
2. WebGPU Crowd Simulation: https://github.com/wayne-wu/webgpu-crowd-simulation/ (参考了一些并行排序算法的 shader 实现)
3. R. Weller, “A Brief Overview of Collision Detection,” in New Geometric Data Structures for Collision Detection and Haptics, R. Weller, Ed., in Springer Series on Touch and Haptic Systems. , Heidelberg: Springer International Publishing, 2013, pp. 9–46. doi: 10.1007/978-3-319-01020-5_2. (参考了碰撞检测的两阶段思路)
4. 用 39 行 Taichi 代码加速 GPU 粒子碰撞检测: https://zhuanlan.zhihu.com/p/563182093 (参考了具体的并行粒子碰撞检测的实现)