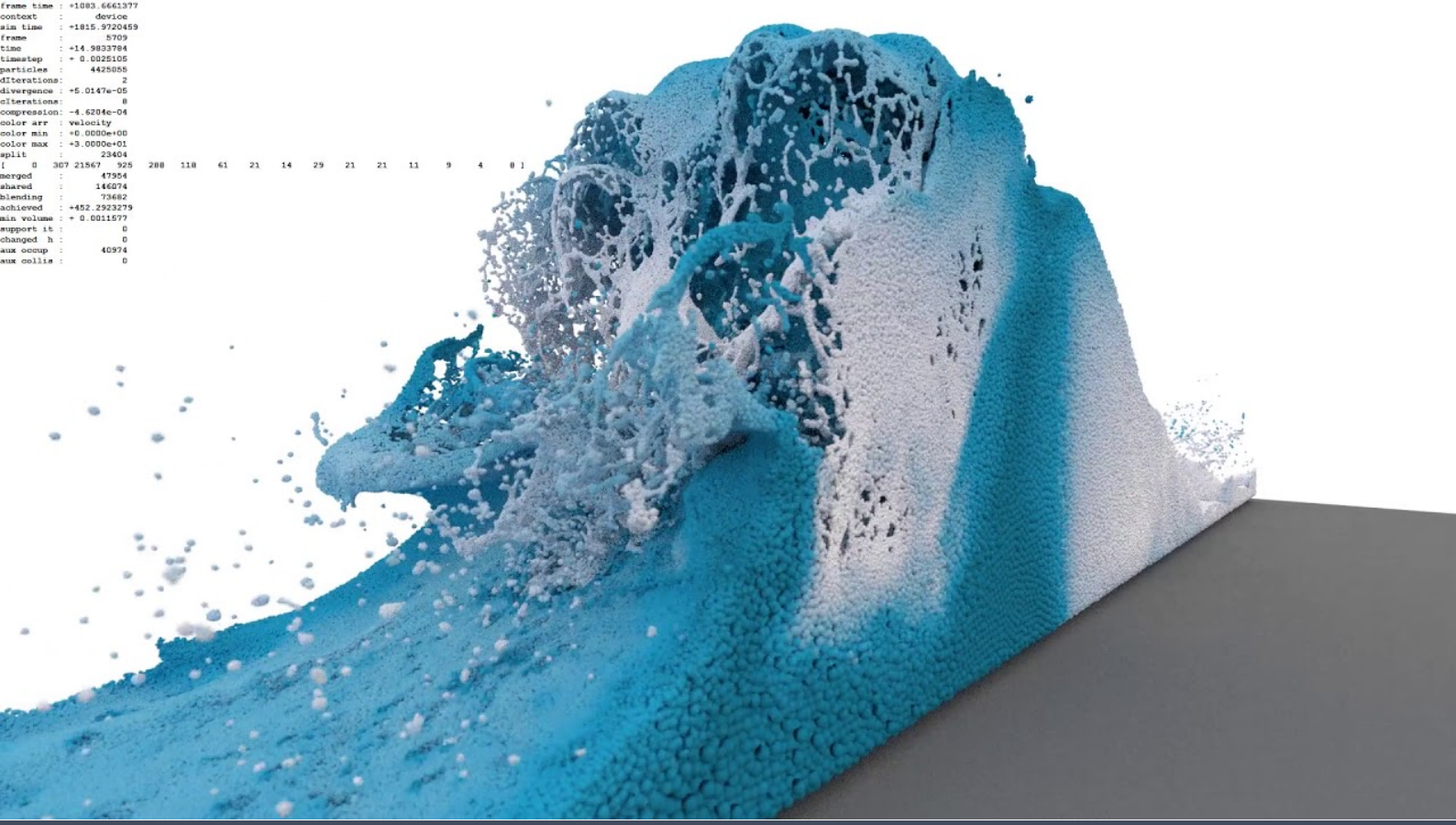

西安交通大学计算机图形学实验文档

小球的运动模拟和碰撞（模拟部分）

作者：罗思源

组织：计算机图形学课题组

时间：November 21, 2022

目录

序

这部分实验将在 Scotty3D 中实现 3D 小球运动的模拟——包含从运动轨迹的计算到简单的碰撞处理。

本文档分为三章：

- **小球的运动学方程**　介绍如何计算小球运动轨迹
- **碰撞的处理**　介绍如何处理小球与墙面的碰撞
- **工程上的实现细节**　在工程层面来指导如何实现上述算法

请大家按顺序认真阅读文档，这可以极大减少实验时遇到的困难。

第 1 章 小球的运动学方程

1.1 基本的运动学规律

当我们去模拟小球的运动时，在每一个时刻下，每个小球都由位置 (position)、速度 (velocity) 和加速度 (acceleration) 来决定当前的状态。因此如何计算出小球的上述三种属性就成了模拟的重要问题。

问题 1.1 运动学方程

$$F = ma \tag{1.1}$$

$$\frac{dv}{dt} = \frac{F}{m} \tag{1.2}$$

$$\frac{dx}{dt} = v \tag{1.3}$$

1.2 时间离散化

除此之外，如何去做离散化本身也是一个重要的问题，自然界中的时间是连续的，但在计算机中时间是被离散的，而我们通常使用 timestep 去表示每一帧的时间，即每一次渲染 & 模拟的时间——每一次重新计算加速度，计算速度，最终传递到位置上的时间。通常为了满足实时的要求，timestep 的设置总是小于 1/30 或 1/60s，以达到 30 帧或者 60 帧的结果。而在求解完运动学方程后，再通过时间离散化来计算出每一个 timestep 下小球的状态并更新，也就完成了一次模拟。

问题 1.2 时间离散化后的离散方程

$$x_{t+\Delta t} = x_t + v_t * \Delta T \tag{1.4}$$

$$v_{t+\Delta t} = v_t + a * \Delta T \tag{1.5}$$

而上述的这些方程都会在 update 函数中来更新 pos 和 velocity 来实现。

注：这种时间离散化的方式实际是存在误差的，也就是我们所说的前向欧拉法 (Forward Euler)，实际上并不满足能量守恒，整个系统的能量变得越来越大。如果同学在实现之后发现小球的速度有越来越大的趋势，是由该因素导致的。但使用隐式方法或者龙格-库塔方法可以有效的避免该问题。但本实验不在这方面做进一步要求，感兴趣的同学可以自己尝试。

更详细的工程方面的介绍请参考 chapter3。

第 2 章 碰撞的处理

2.1 检测碰撞

小球的运动轨迹在求解运动学方程的时候便知晓，而每个 timestep 中小球的运动轨迹其实是一条条射线，即沿当前的速度方向移动 $||velocity|| * dt$ 长度，因此沿当前方向创建一条射线，并在当前步长内查找碰撞。若相交，就可以准确计算出何时可以发生碰撞。

而更细节的则是射线与三角面片求交的工作，可以参考该链接https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersec

2.2 处理碰撞

我们已经有了检测碰撞的手段，并在每一个 timestep 中设置了一个检测碰撞的周期，确保能够检测完全。假设当前 timestep 中已经检测出会发生碰撞。那么我们接下来就要处理碰撞了。

发生碰撞的时候，小球的表现可以调整，比如黏在墙壁上 & 反弹下来。在该实验中，我们的预期表现是小球发生完全弹性碰撞。小球的速度在碰撞前后大小应该相同，但方向应为关于碰撞表面的法线反射方向。

然后再计算发生碰撞之后剩余的 timestep。同时我们也需要考虑在剩下 timestep 中发生碰撞的可能性，因此需要进行重新设置新的碰撞 “timestep”，并再新的 “timestep” 下检测碰撞并处理。

2.3 考虑小球半径

之前的假设中都只是将小球作为粒子来考虑，并未考虑小球半径，但为了让仿真更加逼真，在检测和处理碰撞的时候应该考虑到小球的半径，寻找小球在当前运动射线方向与表面的最小距离，而非小球中心到表面的距离是一个很好的思路。

笔记 小球寿命

小球的寿命由 update() 的返回值控制，若为 false 则删除小球。而每个小球的 age 参数则控制着小球的存活时间。

第 3 章 工程上的实现细节

该 chapter 简单介绍一下工程中需要实现的部分，更详细的版本会在后续习题课中讲解。

3.1 模型导入

运行完 Scotty3D 程序后，在 import object 中导入 “Scotty3D/media/particles.dae” 作为测试工程。没有实现小球模拟前，并没有现象。实现之后则会看到粒子发射器发射的小球运动并与墙面发生碰撞并反弹。

3.2 小球的位置更新

小球的位置更新与碰撞处理都在 “Scotty3D/src/student/particles.cpp” 中的 update() 函数中实现，即见下：

```
#include "../scene/particles.h"
#include "../rays/pathtracer.h"

bool Scene_Particles::Particle::update(const PT::Object& scene, float dt, float radius) {

   // TODO(Animation): Task 4

   // Compute the trajectory of this particle for the next dt seconds.

   // (1) Build a ray representing the particle's path if it travelled at constant
       velocity.

   // (2) Intersect the ray with the scene and account for collisions. Be careful when
       placing
   // collision points using the particle radius. Move the particle to its next position.

   // (3) Account for acceleration due to gravity.

   // (4) Repeat until the entire time step has been consumed.

   // (5) Decrease the particle's age and return whether it should die.

   return false;
}
```

该函数共有两个参数，dt 代表时间步长 (timestep)，radius(半径)，分别指导模拟和碰撞的过程。

其他小球相关类可以参考 “Scotty3D/src/scene/particles.h”，其中：

```
   struct Particle {

      Vec3 pos;
      Vec3 velocity;
      float age;
```

```
    static const inline Vec3 acceleration = Vec3{0.0f, -9.8f, 0.0f};

    bool update(const PT::Object& scene, float dt, float radius);
};
```

分别储存了小球的信息，也意味着在 update 中其实就是更新上述类中所储存的信息。

在 update() 中的实现步骤有如下：

1. 我们需要通过 dt 设置一个整体一帧模拟时间，再人为指定一个碰撞检测精度时间，我们这里推荐同学们设置为 1e-3；

2. 进行模拟和碰撞检测。首先我们可以依据当前的 position 和 velocity 使用 Ray() 来创建一条射线，然后根据 dt 来计算该帧模拟后小球的坐标。再进行碰撞检测（该处可以使用 hit 函数来计算）来检测这一帧中是否发生了碰撞；

3. 如果发生碰撞，则需要计算当前位置到碰撞所需要时间，并计算出碰撞时刻的位置和速度的改变，并计算剩余的模拟时间，并在剩余的模拟时间中检测是否发生碰撞。如果没有发生碰撞，则需要使用 1.1 中所描述的时间离散化方法来模拟；

4. 在粒子寿命的计算中减去 dt。并通过返回值判断粒子是否存活；

因此修改后的 update() 函数见下：

```
#include "../scene/particles.h"
#include "../rays/pathtracer.h"

bool Scene_Particles::Particle::update(const PT::Object& scene, float dt, float radius) {

   float tLeft = dt; // time left for collision looping
   float eps = 1e-3; // minimum time to continue loop

   while((tLeft - eps) > 0) {
      // TODO: ray from particle origin; velocity is always unit
      // TODO: how far the particle will travel

      // TODO: hit something?

      // TODO: if hit something?
      // TODO: calculate new pos and velocity, and new simulation time.

      // TODO: if not hit
         // use Forward Euler to calculate new pos and velocity and break loop

   }

   age -= dt;
   return age > 0; // dead particle?
   return false;
}
```

而碰撞检测的求交部分则是引用了 “Scotty3D/src/student/tri_mesh.cpp” 中的 hit() 函数，这也是引用 “../rays/-pathtracer.h” 头文件的原因。

下面是 hit 的实现方法：

hit 实际计算的是射线和三角形求交，输入是 ray 和三角形的三个顶点，返回值是一个 Trace 类型，在 rays/-trace.h 中定义。若不发生相交，则返回 trace 类型中定义为不相交，若发生相交，则计算相交的位置和法向量。

在 hit() 函数的实现中，该实验仅要求满足 hit 相交时的关于返回 trace 类的各参数计算，包括 origin, hit, distance, position, normal。

因此修改后的 hit 函数见下：

```
Trace Triangle::hit(const Ray& ray) const {

   // Each vertex contains a postion and surface normal
   Tri_Mesh_Vert v_0 = vertex_list[v0];
   Tri_Mesh_Vert v_1 = vertex_list[v1];
   Tri_Mesh_Vert v_2 = vertex_list[v2];
   (void)v_0;
   (void)v_1;
   (void)v_2;

   // TODO (PathTracer): Task 2
   // Intersect the ray with the triangle defined by the three vertices.

   Vec3 e1 = v_1.position - v_0.position;
   Vec3 e2 = v_2.position - v_0.position;
   Vec3 s = ray.point - v_0.position;

   Trace ret;
   float denominator = dot(cross(e1, ray.dir), e2);
   if(denominator == 0) {
      return ret;
   }

   float u = -1 * dot(cross(s, e2), ray.dir) / denominator;
   float v = dot(cross(e1, ray.dir), s) / denominator;
   float t = -1 * dot(cross(s, e2), e1) / denominator;

   if(u >= 0 && u <= 1 && v >= 0 && (v <= 1) & (u + v <= 1)) {
      if(t >= ray.dist_bounds.x && t <= ray.dist_bounds.y) {
         // calculate each parameter for Trace::ret
         return ret;
      }
   }
   return ret;
}
```

参考文献

[1] CMU Graphics. *Scotty3D's Developer Manual: Particles*. URL: https://cmu-graphics.github.io/Scotty3D/animation/particles.