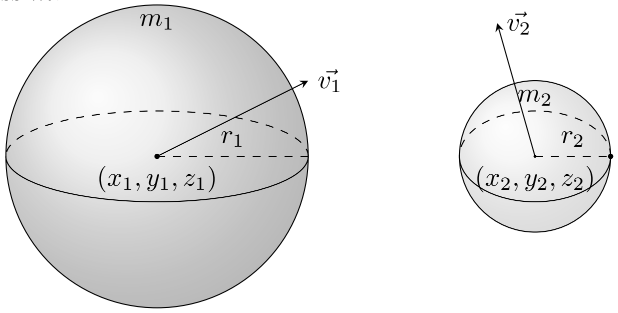
Literate Classical Physics

Tobi R. Lehman

February 13, 2022

1 Introduction

This program simulates a 3D universe subject to the laws of Newtonian mechanics. The basic ontology[1] is a collection of rigid bodies, shaped like spheres, each with a position (x, y, z), a velocity $\vec{v}$, radius r, and a mass m.



The collection of all bodies in the simulation is called the Universe. The state of the Universe is the collection of positions and velocities of all the bodies. The state is updated at each time step t. The state update rule is based on the law of Newtonian gravity:

$$\vec{F_{12}} = G\frac{m_1 m_2}{r^2}\hat{r}$$

To understand how this law applies, remember that force is proportional to acceleration, and acceleration is the rate of change of velocity. In vector form: $\vec{F} = m\vec{a}$, and $\vec{a} = \frac{d}{dt}\vec{v}$. Now that we have connected the force law to the state of the Universe, we can guess how to update the state and compute the next moment. We take all pairs of bodies (i, j) and calculate the forces between them $\vec{F_{ij}}$

¶ Newtonian physics introduced the idea that the force on any body is the sum of all the forces acting on it. Given an n-body system, take the ith body, sum over all the $n - 1$ other bodies' force on i.

$$\underbrace{\sum_j \vec{F_{ij}}}_{\text{sum of all forces acting on the i-th body}} \qquad = m_i\vec{a}_i$$

2 ⟨ Loop over all bodies, add up forces and apply the force to the body 2 ⟩ ≡

```
for (int i = 0; i < n; i++) {
  body *a = &bodies[i];
  vec3 force_on_a = {0, 0, 0};
  for (int j = 0; j < n; j++) {
    body *b = &bodies[j];
    if (i ≠ j) {
      vec3 force_from_b_on_a = {0, 0, 0};
      force_between(a, b, &force_from_b_on_a);
```

[1]An ontology is a scheme defining what exists.

```
          vec3_add(&force_on_a, &force_from_b_on_a);
        }
      }
      ⟨Apply the force to the body a 10⟩;
    }
```
This code is used in chunk 8.

¶ This is the overall structure of the program `lcp.c`

3
```
  ⟨Header files 12⟩
  ⟨Constants 9⟩
  ⟨Struct types 4⟩
  ⟨Distance function 5⟩
  ⟨Function definitions 11⟩
  ⟨The main program 6⟩
```

¶ We need *C* structs that represent the values of all the state variables we are tracking.

The *vec3* type is a triple of real numbers, represented by IEEE 754 floating pointer numbers. I am choosing to overload the notion of "point" and "vector", and use vectors to represent points. For Newtonian physics this works, in Relativity, there is a distinction between points and 4-vectors, but this program is dedidedly non-relativistic.

A *body* has an *position*, a *velocity*, a radius *r*, and a mass *m*.

4 ⟨Struct types 4⟩ ≡
```
  typedef struct vec3 {
    float x, y, z;
  } vec3;
  typedef struct body {
    vec3 position;
    vec3 velocity;
    float mass;
    float radius;
  } body;
```
This code is used in chunk 3.

¶ The *distance* function is essential to calculating the force between two bodies.

5 ⟨Distance function 5⟩ ≡
```
  float distance(body *a, body *b)
  {
    float dx = a→position.x − b→position.x;
    float dy = a→position.y − b→position.y;
    float dz = a→position.z − b→position.z;
    return sqrtf(dx * dx + dy * dy + dz * dz);
  }
```
This code is used in chunk 3.

¶ Here is the general layout of the *main* function.

6 ⟨The main program 6⟩ ≡
```
int main()
{
  ⟨Set up initial conditions of universe 7⟩;
  ⟨The main time loop 8⟩;
}
```
This code is used in chunk 3.

¶ The initial conditions of the universe are the set of bodies, their positions, masses and velocities. The laws of physics and the inexorable march of time takes over after that. Let's start with a binary star system. The **body**[] type is an array of **body**s, the two bodies in this example are two stars orbiting each other. We make use of the fact that two equal mass bodies will form a stable binary orbit of their speed is $Gm/4r$ and they are distance r apart, each traveling in opposite directions:

7 ⟨Set up initial conditions of universe 7⟩ ≡
```
int n = 2;
float m = 10.0;
float r = 2;
float v = sqrtf((G * m)/(4 * r));
body bodies[] = {{{-1, 0, 0}, {0, v, 0}, m, 1}, {{1, 0, 0}, {0, -v, 0}, m, 1}};
```
This code is used in chunk 6.

¶ The main time loop is where the simulated time flows. Each iteration of the loop adds dt seconds to the current time.

8 ⟨The main time loop 8⟩ ≡
```
for (float t = 1.0; t < 1000.0; t += dt) {
  ⟨Loop over all bodies, add up forces and apply the force to the body 2⟩;
  ⟨Output the state 14⟩;
}
```
This code is used in chunk 6.

¶ Physics has many constant values, like the speed of light c, the gravitational constant G. In this humble program, there is another constant, dt, the minimum number of seconds used to advance the time loop. For practical reasons, this is much larger than the Planck length.

9 ⟨Constants 9⟩ ≡
```
const float dt = 0.00001;
const int G = 1;
```
This code is used in chunk 3.

¶ Now that *force_on_a* has been calculated, we use Newton's equations: $\vec{F} = m\vec{a}$, and the fact from calculus that $\vec{a} = \frac{d}{dt}\vec{v}$ to update the velocity of the body a. The change in acceleration of the body a over the dt time step is equal to $\vec{F}/m$ where m is the mass of the body a.

10 ⟨Apply the force to the body a 10⟩ ≡
```
float m = a->mass;
a->velocity.x += (force_on_a.x/m) * dt;
a->velocity.y += (force_on_a.y/m) * dt;
a->velocity.z += (force_on_a.z/m) * dt;
a->position.x += a->velocity.x * dt;
a->position.y += a->velocity.y * dt;
a->position.z += a->velocity.z * dt;
```
This code is used in chunk 2.

¶ In a previous section we used a few functions we haven't defined yet, one calculates the graviational force between two bodies, the other does vector addition. In the typical C style, we pass pointers to our structs as a way to get return values.

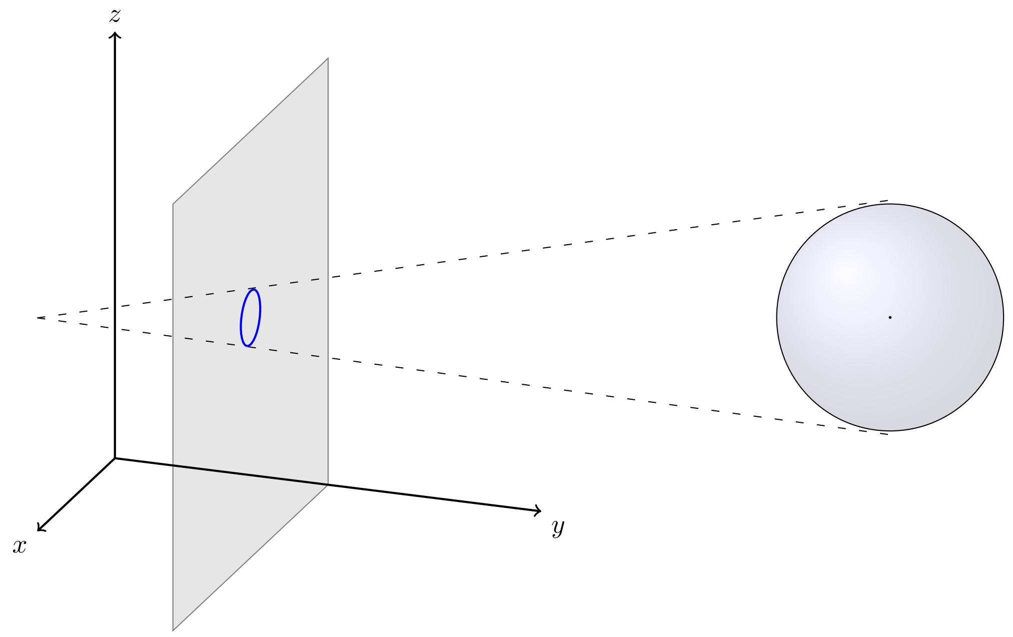
11 ⟨Function definitions 11⟩ ≡
```
void force_between(body *a, body *b, vec3 *f)
{
  float m_a = a→mass;
  float m_b = b→mass;
  float r = distance(a, b);
  float magnitude = (G * m_a * m_b)/(r * r);
  f→x = magnitude * (b→position.x − a→position.x)/r;
  f→y = magnitude * (b→position.y − a→position.y)/r;
  f→z = magnitude * (b→position.z − a→position.z)/r;
}
void vec3_add(vec3 *output, vec3 *to_add)
{
  output→x += to_add→x;
  output→y += to_add→y;
  output→z += to_add→z;
}
```
This code is used in chunk 3.

¶ We need standard I/O and the math library (remember to compile with `-lm`)

12 ⟨Header files 12⟩ ≡
```
#include <stdio.h>
#include <math.h>
```
This code is used in chunk 3.

2 Rendering

We want the state of the 3D universe to be displayed on a 2D screen. For this we will build a ray tracer. A ray tracer simulates rays of light that reflect off of the 3D scene and then hit the camera.



¶ For debugging purposes, before we implement rendering, we should output the states to stdout:

14 ⟨Output the state 14⟩ ≡

```
printf("(%f,␣%f,␣%f),␣(%f,␣%f,␣%f)\n", bodies[0].position.x, bodies[0].position.y, bodies[0].position.x,
        bodies[1].position.x, bodies[1].position.y, bodies[1].position.x);
```

This code is used in chunk 8.

¶ TODO: start working on a ray tracer

Index

List of Refinements

⟨Apply the force to the body a 10⟩ Used in chunk 2.
⟨Constants 9⟩ Used in chunk 3.
⟨Distance function 5⟩ Used in chunk 3.
⟨Function definitions 11⟩ Used in chunk 3.
⟨Header files 12⟩ Used in chunk 3.
⟨Loop over all bodies, add up forces and apply the force to the body 2⟩ Used in chunk 8.
⟨Output the state 14⟩ Used in chunk 8.
⟨Set up initial conditions of universe 7⟩ Used in chunk 6.
⟨Struct types 4⟩ Used in chunk 3.
⟨The main program 6⟩ Used in chunk 3.
⟨The main time loop 8⟩ Used in chunk 6.