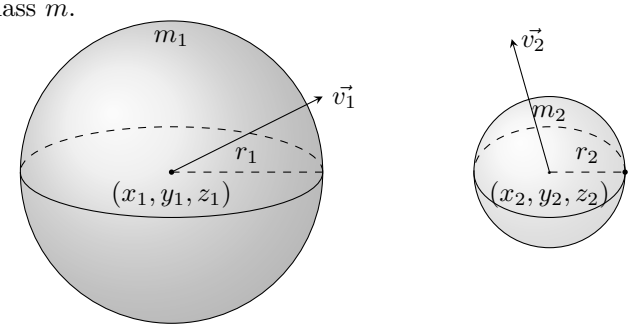
Literate Classical Physics

Tobi R. Lehman

February 16, 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.

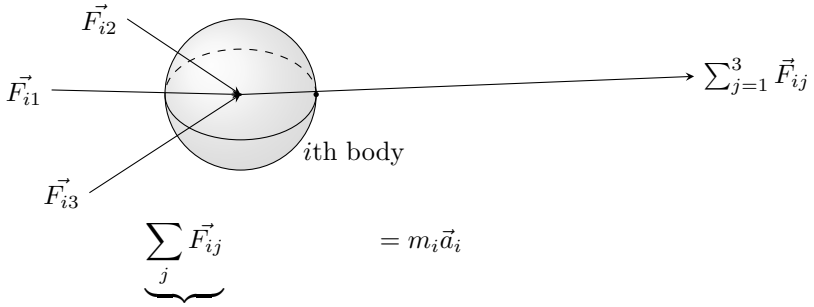


The collection of all bodies in the simulation is called the Universe. The state of the Universe is the collection of positions and velocities of all the bodies. The state is updated at each time step t. The state update rule is based on the law of Newtonian gravity:

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

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. To calculate the total force on the ith body, in an n-body system, sum over all the $n-1$ other bodies' force on i.



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

¶ From the above section, we can see how to calculate the force on each body at any given moment in time. This loop finds the instananeous snapshot of all forces in the simulation. These forces will then be applied to the bodies to update their velocities and positions.

3 ⟨ Loop over all bodies, add up forces and apply the force to the body 3 ⟩ ≡

```
for (int i = 0; i < number_of_bodies; i++) {
  body * ith_body = &bodies[i];
  vec3 force_on_ith_body = {0, 0, 0};
  for (int j = 0; j < number_of_bodies; j++) {
    body * jth_body = &bodies[j];
    if (i ≠ j) {
      vec3 force_from_j_on_i = {0, 0, 0};
      force_between(ith_body, jth_body, &force_from_j_on_i);
      vec3_add(&force_on_ith_body, &force_from_j_on_i);
    }
  }
  ⟨ Apply the force to the body 4 ⟩;
}
```

This code is used in chunk 10.

¶ Now that *force_on_ith_body* 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 ith body. The change in acceleration of the ith body over the dt time step is equal to $\vec{F}/m$ where m is the mass of the body.

4 ⟨ Apply the force to the body 4 ⟩ ≡

```
float m = ith_body->mass;
ith_body->velocity.x += (force_on_ith_body.x/m) * dt;
ith_body->velocity.y += (force_on_ith_body.y/m) * dt;
ith_body->velocity.z += (force_on_ith_body.z/m) * dt;
ith_body->position.x += ith_body->velocity.x * dt;
ith_body->position.y += ith_body->velocity.y * dt;
ith_body->position.z += ith_body->velocity.z * dt;
```

This code is used in chunk 3.

¶ When applying the force, we update the state. Here is where we define the C structs that store that state.

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 decidedly non-relativistic.

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

5 ⟨ Struct types 5 ⟩ ≡

```
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 6.

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

6 ⟨ Header files 13 ⟩
⟨ Constants 11 ⟩
⟨ Struct types 5 ⟩
⟨ Rendering function 16 ⟩
⟨ Distance function 7 ⟩
⟨ Function definitions 12 ⟩
⟨ The main program 8 ⟩

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

7 ⟨ Distance function 7 ⟩ ≡
```
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 6.

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

8 ⟨ The main program 8 ⟩ ≡
```
int main()
{
  ⟨ Set up initial conditions of universe 9 ⟩;
  ⟨ The main time loop 10 ⟩;
}
```
This code is used in chunk 6.

¶ 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:

9 ⟨ Set up initial conditions of universe 9 ⟩ ≡
```
int number_of_bodies = 2;
float mass = 10.0;
float radius = 2;
float v = sqrtf((G * mass)/(4 * radius));
body bodies[] = {{{-1, 0, 0}, {0, v, 0}, mass, 1}, {{1, 0, 0}, {0, -v, 0}, mass, 1}};
```
This code is used in chunk 8.

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

10 ⟨ The main time loop 10 ⟩ ≡
```
for (float t = 1.0; t < 1000.0; t += dt) {
  ⟨ Loop over all bodies, add up forces and apply the force to the body 3 ⟩;
}
```
This code is used in chunk 8.

¶ 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.

11 ⟨ Constants 11 ⟩ ≡

```
const float dt = 0.00001;
const int G = 1;
```

See also chunk 15.

This code is used in chunk 6.

¶ 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.

12 ⟨ Function definitions 12 ⟩ ≡

```
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 6.

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

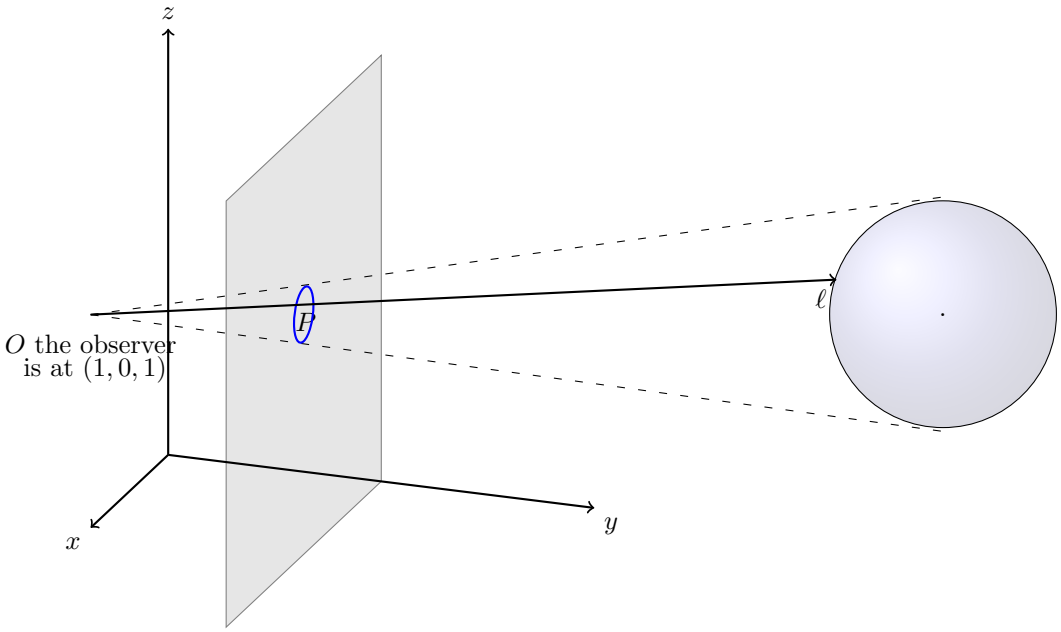
13 ⟨ Header files 13 ⟩ ≡

```
#include <stdio.h>
#include <math.h>
```

This code is used in chunk 6.

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 objects in the 3D scene and then hit the camera.



Light travels along the shortest path through space. In Newtonian mechanics, space is Euclidean, so that means light travels in a straight line. The way to simulate light that hits the 2D screen is to pick a point that corrsponds to a pixel, then trace that light backwards to see what objects it bounced off of.

To trace a ray of light, we can take the point O, the observer, and the point P that is mapped to the pixel. Let ℓ be the line defined by O and P. If ℓ does not intersect any bodies, then color that pixel black. Otherwise, color the pixel based on the surface color and the distance from O (farther away is assumed to be darker).

The line ℓ can be parameterized like this $\vec{\ell}(s) = \vec{O} + s\hat{\ell}$. The surface of the ith spherical body is described by the inequality $(\vec{x} - \vec{C}) \cdot (\vec{x} - \vec{C}) < r^2$. We can substitute the line equation into the sphere and solve the resulting quadratic to find if it intersects or not.

14 ⟨ Trace ray of light incident on pixel 14 ⟩ ≡
```
TODO
```
This code is used in chunk 16.

¶ Before we can define the pixel space, we need to choose a pixel *width* and *height*.

15 ⟨ Constants 11 ⟩ +≡
```
const int pixel_width = 800;
const int pixel_height = 600;
```

¶ Pick a pixel (p_i, p_j), then calculate the line from the observer O through $(p_i, 1, p_j)$, which is the 3D position of the pixel on the gray 2D screen.

16 ⟨ Rendering function 16 ⟩ ≡
```
void render(body bodies[])
{
  for (int i = 0; i < pixel_width; i++) {
    for (int j = 0; j < pixel_height; j++) {
      ⟨ Trace ray of light incident on pixel 14 ⟩;
```

```
            ⟨ Store color for pixel 17 ⟩;
        }
    }
}
```

This code is used in chunk 6.

¶ Now that we have calculated and stored all the pixel colors, we want to display them. For the initial version of the program, we will use the excellent stb libraries to write a BMP file as output.

17 ⟨ Store color for pixel 17 ⟩ ≡

```
TODO
```

This code is used in chunk 16.

¶ Next step:

Index

a: 7, 12.
b: 7, 12.
bodies: 3, 9, 16.
body: 3, 5, 7, 9, 12, 16.
distance: 7, 12.
dt: 4, 10, 11.
dx: 7.
dy: 7.
dz: 7.
f: 12.
force_between: 3, 12.
force_from_j_on_i: 3.
force_on_ith_body: 3, 4.
G: 11.
height: 15.
i: 3, 16.
ith_body: 3, 4.
j: 3, 16.
jth_body: 3.
m: 4.
m_a: 12.
m_b: 12.
magnitude: 12.
main: 8.
mass: 4, 5, 9, 12.
number_of_bodies: 3, 9.
output: 12.
pixel_height: 15, 16.
pixel_width: 15, 16.
position: 4, 5, 7, 12.
r: 12.
radius: 5, 9.
render: 16.
sqrtf: 7, 9.
t: 10.
to_add: 12.
`TODO`: 14, 17.
v: 9.
vec3: 3, 5, 12.
vec3_add: 3, 12.
velocity: 4, 5.
width: 15.
x: 5.
y: 5.
z: 5.

List of Refinements

⟨ Apply the force to the body 4 ⟩ Used in chunk 3.
⟨ Constants 11, 15 ⟩ Used in chunk 6.
⟨ Distance function 7 ⟩ Used in chunk 6.
⟨ Function definitions 12 ⟩ Used in chunk 6.
⟨ Header files 13 ⟩ Used in chunk 6.
⟨ Loop over all bodies, add up forces and apply the force to the body 3 ⟩ Used in chunk 10.
⟨ Rendering function 16 ⟩ Used in chunk 6.
⟨ Set up initial conditions of universe 9 ⟩ Used in chunk 8.
⟨ Store color for pixel 17 ⟩ Used in chunk 16.
⟨ Struct types 5 ⟩ Used in chunk 6.
⟨ The main program 8 ⟩ Used in chunk 6.
⟨ The main time loop 10 ⟩ Used in chunk 8.
⟨ Trace ray of light incident on pixel 14 ⟩ Used in chunk 16.