PA5

Gravity simulation

FORC - Forritun í C++, spring 2022
Reykjavik University - School of Computer Science, Menntavegi 1, IS-101 Reykjavík, Iceland

Arnór Þorleifsson
arnort20@ru.is
Ragnar Smári Ómarsson
ragnaro20@ru.is
Þorsteinn I Stefánsson Rafnar
thorsteinns20@ru.is

May 10, 2022

1 Introduction

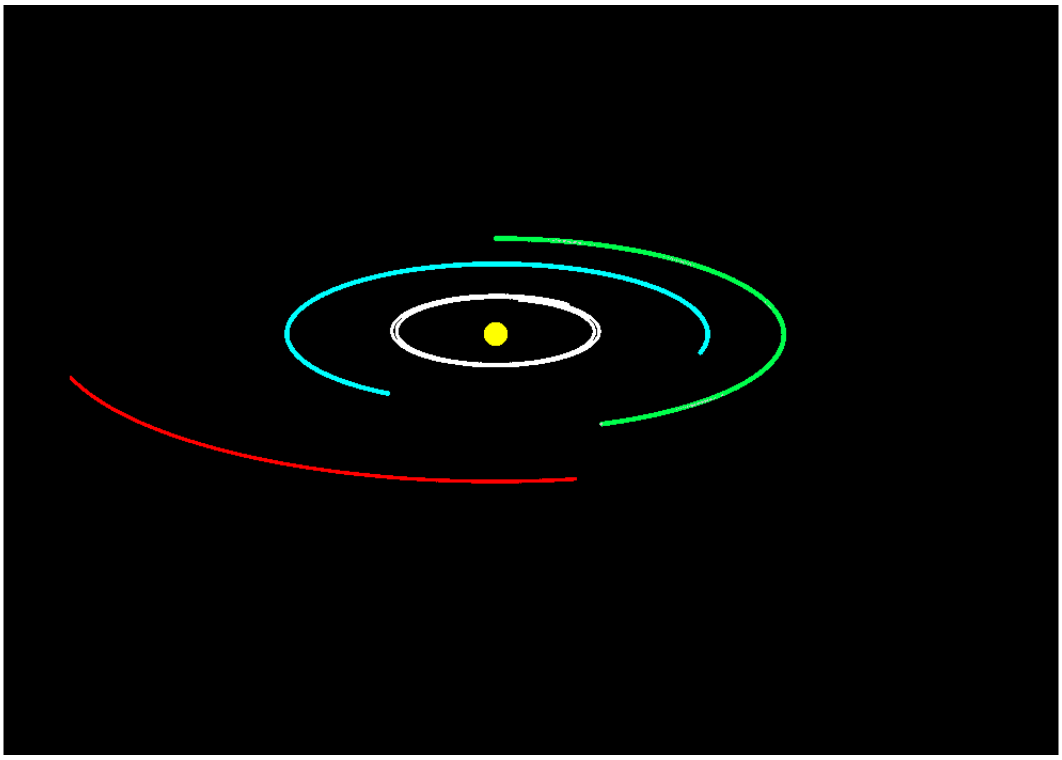
Figure 1: Tilted with trail turned on

Ever since the Italian astronomer Galileo Galilei looked upon the night sky people have wondered about the journey that these celestial bodies take. In this assignment we attempt to model this in a visually appealing and interactive way. Our gravity simulation tool uses OpenGL to display the planets, whose position, velocity and acceleration are calculated using multithreading in C++.

2 It's not rocket science
Oh wait, yes it is!

In order to get the physics simulation working at all, we had to come up with the math and the means to compute all that math in C++.
Newton's law: $F = G * \frac{M_1 * M_2}{r^2}$ 9.1
Once we've calculated the gravitational pull on the planet, we get the angle towards the mass that's acting on it and translate it to an acceleration on the planet's x and y coordinates.9.2

3 First iterations

The first iteration of the project involved only the earth as a planet and the sun was defined as a mass in the center of the solar system that was not affected by gravity.
This served as a proof of concept to refine the physics equations until we got the earth to go around the sun in a stable orbit.
The sun was then changed to a planet and the other planets of the solar system were added one at a time. All the values given to each planet were taken from the official NASA website and converted to SI units.

4 Multi-threading

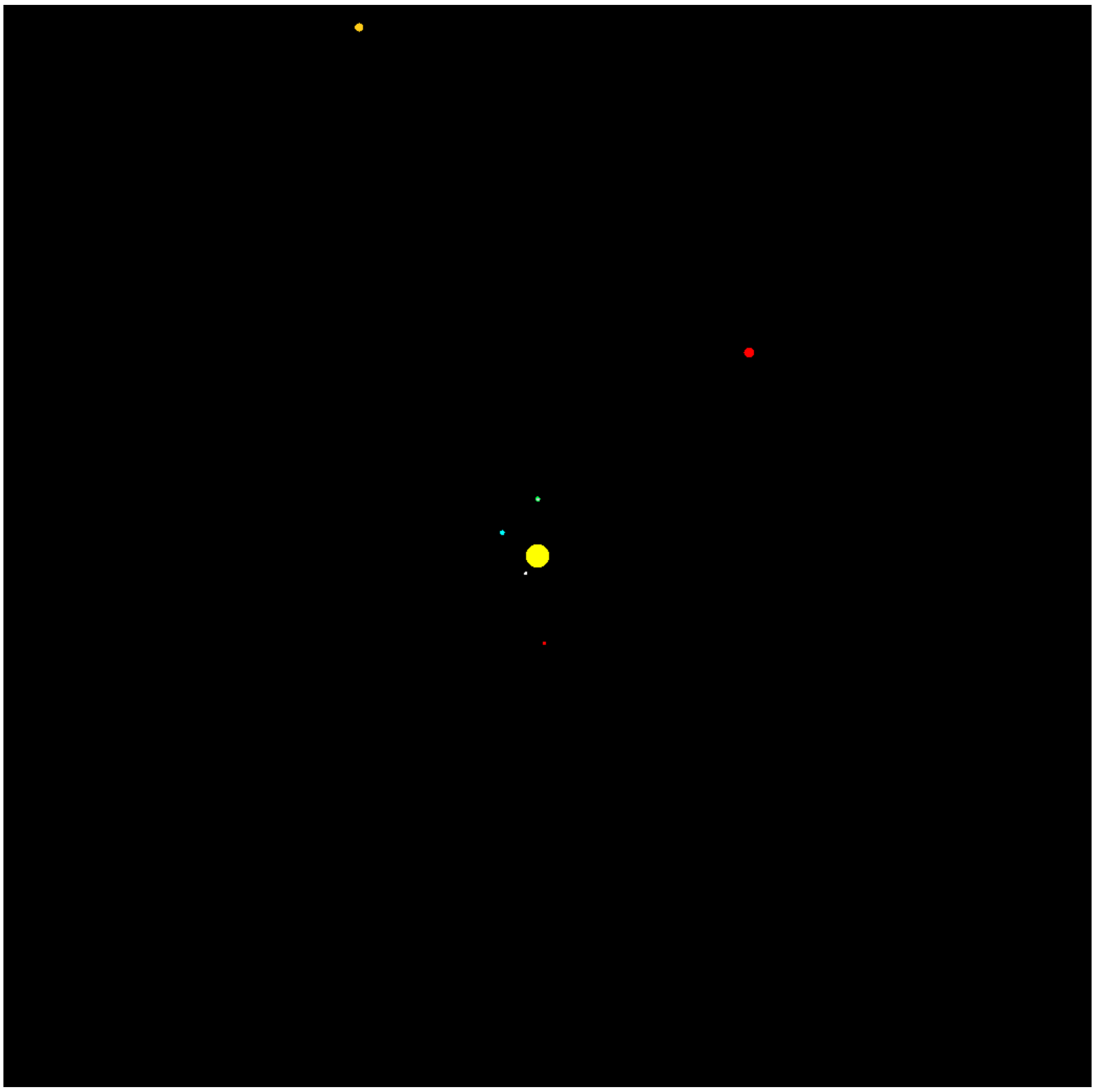

Figure 2: Wide view with no trail

The main problem after adding all the planets in the solar system was that this simulation had to calculate the force acting on each planet by all the other planets. To speed up the calculations, we implemented multi-threading to have each planet's force calculation its own thread. 9.3
A key detail is that the planets coordinates and velocity is not updated instantly, during the threaded calculations. Only it's acceleration vector is incremented. This makes sure that the data that the threads use remain untouched until all calculations have been made. Finally, when the planets positions and velocities are updated, the acceleration x and y values are reset to zero.

5 Interactivity

We wanted our program to have some interactivity. For simplicity's sake we decided to use keyboard inputs for this, so we defined our interface, shown in the terminal during runtime.

5.1 Add a stable planet

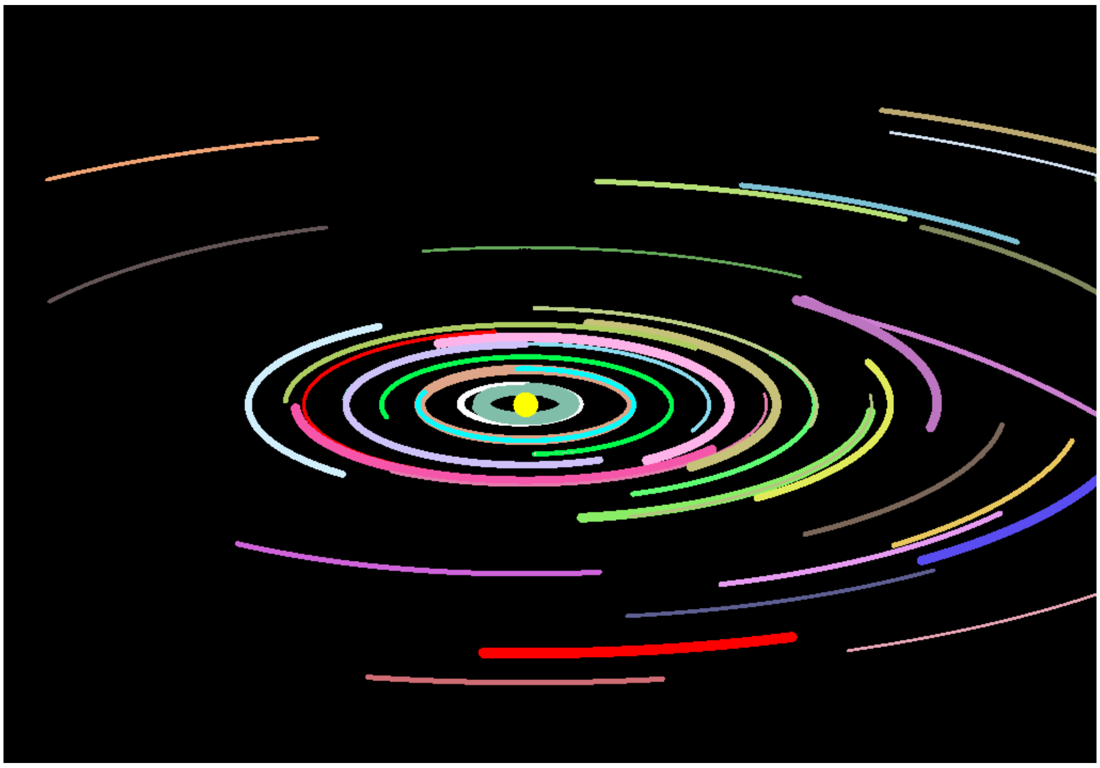

Figure 3: The solar system with a few extra planets

Generating a random planet with a stable orbit is fairly simple, using polar coordinates, which are defined by a distance from the center and an angle. The formula for a stable orbital velocity is as follows:

$$v = \sqrt{\frac{G \cdot M_c}{R}}$$

Where M_c = Mass of center object
G = Gravitational constant
R = distance between the center of the two masses

5.2 Add a rouge planet

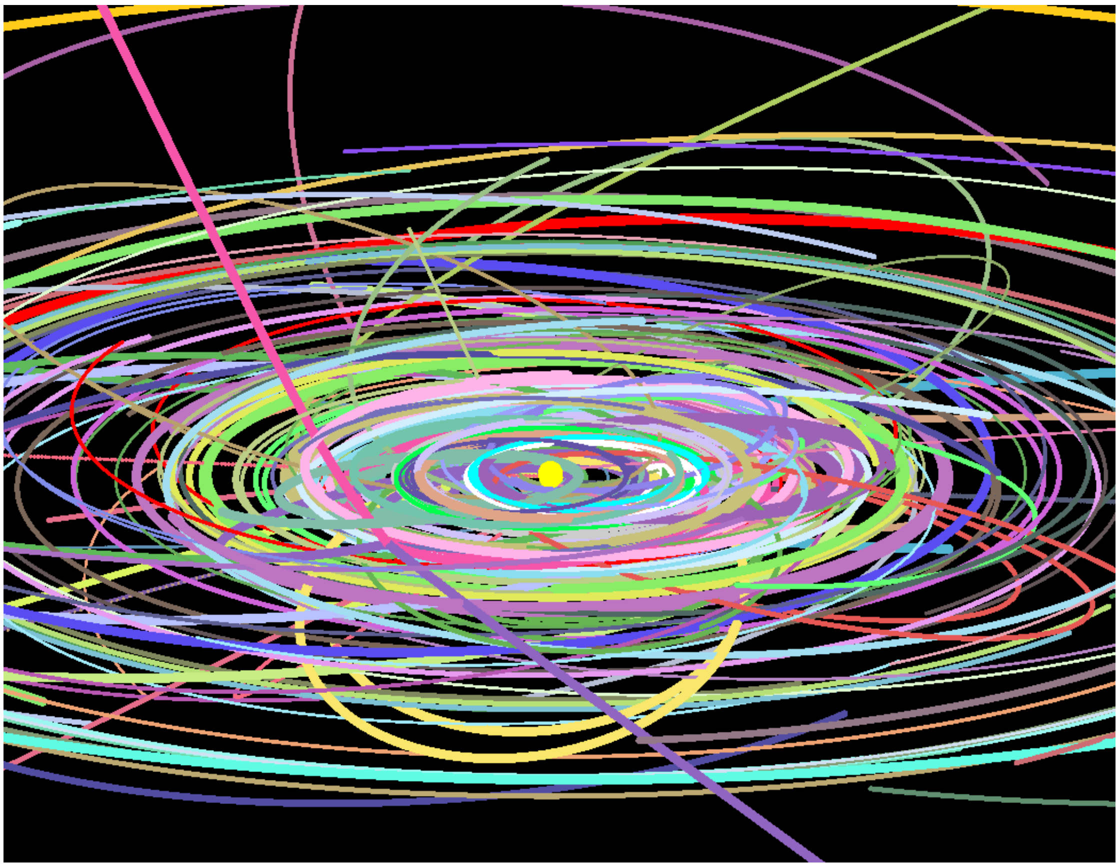

Figure 4: The solar system with a bunch of rouge planets, and some stable ones

Rouge planets do not use any equations to get a correct orientation or speed. They just go somewhere.

5.3 Adding a mass as a terminal command

If you want to create a custom made planet within the solar system, you can use our super-duper user friendly terminal command to make whatever you want appear in the solar system!

Example inputs can be found in the readme file.

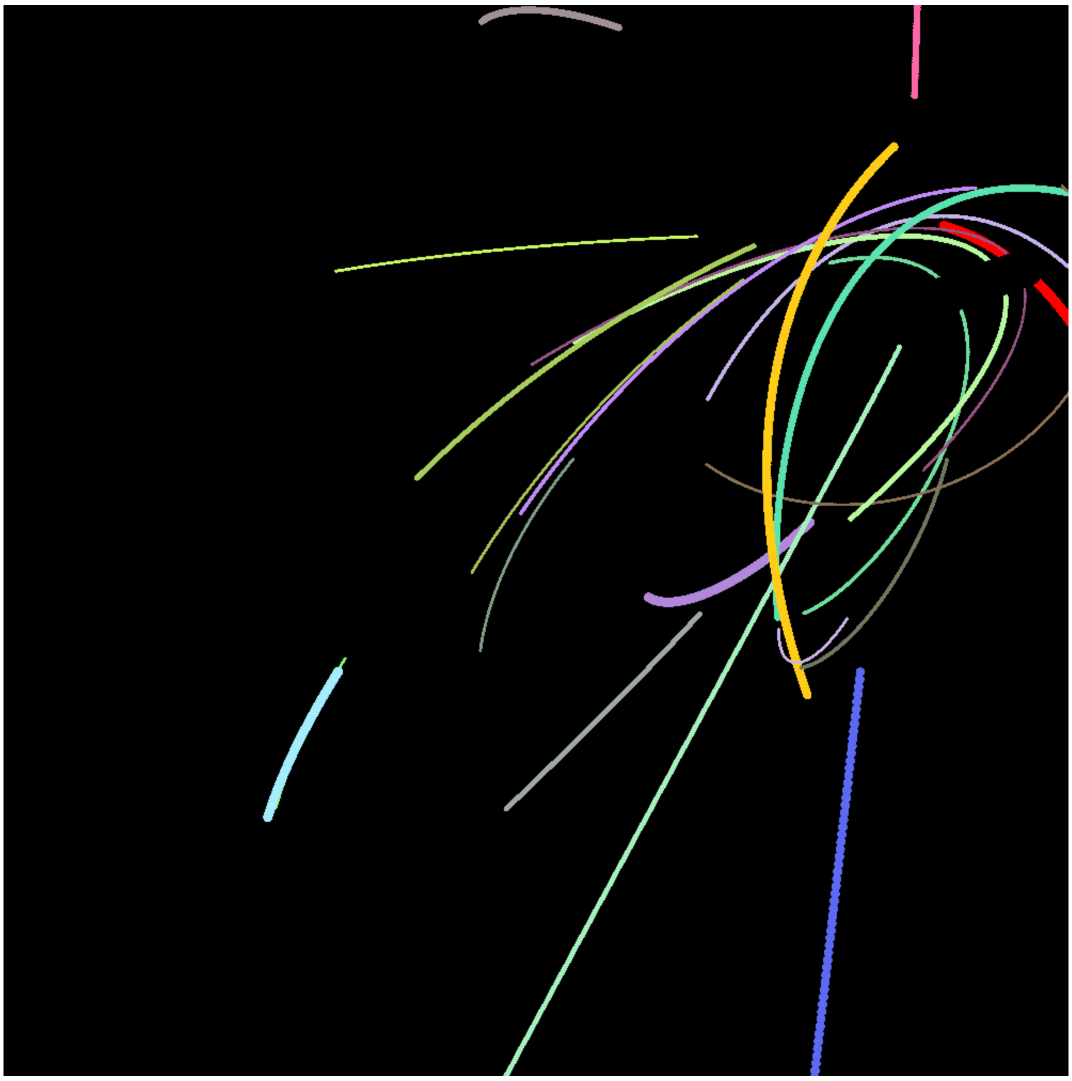

Figure 5: What if a black hole randomly popped up in the solar system?

6 Problems encountered

While starting the project, we were first going to use OpenCV. This proved too difficult to install and start coding and so we tried to find a new graphics library. The next option was graphics.h, this had even more issues as not a lot of people use it and therefore there is little documentation on how to use it. It was also a pain to install and needed a special GCC distribution to work, this would've been too much work for the TA. We eventually decided to go with OpenGL.

7 Conclusion

While the project had its ups and downs we feel that the end result goes above and beyond our initial expectations. We feel that based on the work put into it and the quality of the final product is worth the full 100 points, if not more. Below is an assessment of fulfillment of our project proposal which we made as a scale of 0 to 110 due to the ambitious nature of the project:

- Interface with interactive window: Our program opens up a window, visually representing the solar system. It can be interacted with using the keyboard. 30/30
- Make program take simple decisions: The program takes quite a bit more than simple decision. Due to extensive complexity, for example generating a random planet with a stable orbit, this deserves bonus points. 30/20
- Compile as library: The expectation was to compile it using a makefile, which we did. We do include a library file in the submission but we are not entirely sure if/how it works. 5/20
- Use threads with proper concurrency management: We use threads to calculate the acceleration for each planet relative to the others. This was quite complex to get working properly but we did manage it. The threads are then joined and the actual position/velocity update is performed linearly. 45/40

Total: 110/110
We feel that the extra points for some sections is appropriate given that initial expectations were greatly exceeded and we added features (planet generation) that result in a very impressive final product.

8 Future work

This simulator has plenty of room for improvements! Simply by adding the Z axis, we can make this into a rudimentary simulator fit for an astrophysics research lab. You'd need a whole lot more processing power to run the program for any practical research, though.

9 Code references

9.1 Newton's law

```
const GLdouble G = 6.67e-11;
static GLdouble getForce(GLdouble mass1,GLdouble mass2,GLdouble radius){
    return (G * ((mass1 * mass2)/(radius*radius)));
}
```

9.2 Translate force to acceleration vector

```
static struct Coordinate forceVector(GLdouble gravForce, GLdouble angle ){
    struct Coordinate force;
    force.x = gravForce * cos(angle);
    force.y = gravForce * sin(angle);
    return force;
}

static double getAngle(GLdouble x1, GLdouble x2, GLdouble y1, GLdouble y2){
    return atan2(y1 - y2, x1 - x2);
}
GLdouble get_distance(GLdouble x2, GLdouble y2) {
    return sqrt(pow((x - x2), 2) + pow((y - y2), 2));
};

void update_acceleration(vector<Planet*> solarSystem) {
    // Reset acceleration
    this->acc_x = 0.0;
    this->acc_y = 0.0;

    for(auto planet : solarSystem) {
        if (planet != this) {

            // Calculate force and acceleration relative to current planet
            float angle = getAngle(planet->x, this->x, planet->y, this->y);
            GLdouble distance = this->get_distance(planet->x, planet->y);
            GLdouble force = getForce(this->mass, planet->mass, distance);
            struct Coordinate forceVec = forceVector(force, angle);

            // Increment acceleration on x and y axis
            this->acc_x += forceVec.x / this->mass;
            this->acc_y += forceVec.y / this->mass;


        }
    }
}
```

9.3 Multithreading

```
void threaded_update(){
    thread threads[this->allPlanets.size()];
    int i = 0;
    for(auto planet : this->allPlanets){
        vector<Planet*> solar_system = this->allPlanets;
        threads[i] = thread(&Planet::update_acceleration, planet, solar_system)
        i++;
    }
    for(int i=0; i<this->allPlanets.size(); i++){
        threads[i].join();
    }
    for(auto planet : allPlanets) {
        planet->update_position();
    }
}
```

9.4 OpenGL - Creating a window

```
random_device rd;
    default_random_engine eng(rd());
    uniform_real_distribution<double> angle_distr(0.0, M_PI*2.0);
    glEnable(GL_COLOR_MATERIAL);
    GLFWwindow* window;
    glfwSetErrorCallback(error_callback);
    if (!glfwInit())
        exit(EXIT_FAILURE);
    window = glfwCreateWindow(1080, 1080, "The edge of space", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);
```

9.5 OpenGL - Running the window

```
 while (!glfwWindowShouldClose(window))
    {
        glfwGetFramebufferSize(window, &width, &height);
        u->draw();
        u->threaded_update();
        glViewport(0, 0, width, height);
        glfwSwapBuffers(window);
        glfwPollEvents();
        glFlush();
        //this_thread::sleep_for(std::chrono::milliseconds(100));

    }
    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
```