# Newton's World

https://github.com/michael-s-teodor/newtons-world

Mihnea S. Teodorescu & Andrei Dumitriu
michael.s.teodor@gmail.com & a.dumitriu@student.rug.nl

December 14, 2019

## 1 Project description

This minimalist physics engine is able to compute the trajectories of three-dimensional objects in accordance with Newton's laws of motion. The program features a graphical interface which enables the user to observe the simulation in real time and to adjust its run speed and time step. An upcoming version of the program shall also enable the user to alter the objects' properties (e.g. to remove the Sun out of the Solar System or to double the Earth's mass) at any given state of the process.

## 2 Problem analysis

In order to achieve reasonably precise results, but also to maintain the simplicity of the numerical model, we interpret the problem as follows: every object is always attracted to each other object in accordance with Newton's law of universal gravitation. Furthermore, two objects colliding will always result in the merging of the two into a new object of summed-up mass and volume. This interpretation is motivated by the following statements:

- Orbital precision: Disregarding phenomena such as Mercury's orbit – which follows a path through a region of the Solar System where spacetime is strongly disturbed by the Sun's mass – Newtonian mechanics can be regarded to accurately enough predict the evolution (or history) of any given solar-analogue-based stellar system.
- Collisional imprecision: Apart from having to sum up rather ridiculous velocities, the collision of celestial bodies is very likely to also result in the scattering of matter of different masses and momenta – parts of which may even escape the bodies' gravitational field. However, building a realistic object collider was not the goal of this project's current phase. Moreover, imprecision might actually be decreased by simply merging any colliding objects instead of generating a random collision.
- Efficiency: The required computing power shall also be reduced significantly with this simplified approach.

From Newton's law of universal gravitation we get that $F_g = G \cdot \frac{m_1 \cdot m_2}{r^2}$, where $F_g$ is the gravitational force between two objects of masses $m_1$ and $m_2$ and $r$ is the centre-to-centre distance between them. Moreover, in order to achieve the above-mentioned results in the event of collision, the following equations will be used for merging any two colliding bodies: $\vec{v}_{new} = \frac{\vec{v}_1 * m_1 + \vec{v}_2 * m_2}{m_1 + m_2}$ for the object's new velocity, $m_{new} = m_1 + m_2$ for its new mass, and $r_{new} = \sqrt[3]{{r_1}^3 + {r_2}^3}$ for its new radius.

## 3 Design

This section will cover in depth the two main components of *Newton′s World*: the physics engine, represented by the script *Physics.cs*, and the user interface, constituted by every other script which helps achieve the user experience.

- The physics engine: This script manages a numerical model of spheres (stored in an array of custom data structures named *SphereObject*). The program has a public time step and a run speed – both configurable by the user anytime within execution. Firstly, the program checks with the function *GetInput* whether there are any inputted commands and acts accordingly. The first step in computing the trajectory of any sphere is to determine the its acceleration towards every other sphere. For this, the function *UpdateAcc* first sets the acceleration of every sphere to 0. Then, it compares every existing sphere with every other existing sphere in order to compute the gravitational force between the two (with the help function *GravForce*) and to compute their acceleration by dividing this to their masses respectively. This results in a time complexity of $\mathcal{O}(n^2)$. Further, the function *UpdateVelPos* ensures each sphere's velocity and position is updated. At that point, the function *MergeCollided* is called to check weather there have been any collisions between objects – should any be found, the two spheres are merged in accordance with the equations previously described. Once again, this adds up to an $\mathcal{O}(n^2)$. Finally, the computed data is passed to the UI through the function *UpdateGameSphere*.

- The user experience: The main menu consists of some text elements, an image and 8 buttons. All of them call certain functions in the MainMenu script, which is attached to the Menu Object. Four of these functions change the scene. One is responsible for activating the Help Panel, which is always in the Menu Scene, but it is inactive until the Help button is pressed. The Ok button is part of the Help Panel and it calls the function in MainMenu that deactivates the entire Panel. There is also the button that mutes/unmutes the music, but we will talk about its associated function a bit later. Finally, there is the Exit button, which is responsible for quitting the application. The idea was to have the music play continuously through scene changes, and once it was muted, it needed to stay muted in the different scenes. This is we have created an object that outputs the audio, with a script that specifically says that it needs to remain active through any and all scene changes. This script, SoundObject, also has a function which sets the truth value of the Audio Source, effectively muting/unmuting it. If the Music Source object that has this script by default, when the user returns to the Menu, it will have this newly created Music Source, on top of any pre-existing ones that got carried over from before. This is why the MainMenu script first checks if there are any objects with the tag Sound Source. If it can not find any, it will create an instance of it. If there was already one, the variable SoundObject (of the GameObject type) becomes the already existing Music Source. In the function associated with the Music button, we have a call to the Music Source's function that activates or deactivates the sound. In the other 4 scenes, all of the visible spheres have the InspectSpheres script attached. This script houses all of the variables associated with each sphere. It also updates them on every frame, ensuring that the values are consistent with those in the Physics script. This is also the script that is responsible for instantiating the Planet Inspector (this happens when the user left clicks on the sphere), which is an image with text fields that display the corresponding values. To change these values, the InspectSpheres script calls the functions for each variable on every frame, provided that the sphere's corresponding Inspect Planet is active. The Inspector script, in addition to displaying the values from InspectSpheres, also handles the Ok button, which deactivates the inspector. The values for each sphere are provided either through the Unity Editor, or through the CustomizationScript, in the case of the Custom scene. We have found that in order for the InspectSpheres to properly process these numbers, which will be fed into the Physics script, the NW Manager (the object which has the Physics script) must be placed in the scene after all of the calculations are done. This is not a problem for the Custom scene, because there, NW Manager is spawned after the user clicks on Start, but in every other scene, we use the object Begin Thingy to create an instance of NW Manager 0.5 seconds after the user has entered the scene. One final use of InspectSpheres is to ensure that the spheres are displayed with reasonable numbers. In order to simplify the math in the 3D space, we take the empirical x, y, z positions and we divide them by 1e9, to obtain the positions in 3D space. In the case of the Radius, we found that also dividing it by 1e9 made it problematic to actually view the spheres, especially in the Solar System scene. This is why we are only dividing it by 1e7 to obtain the scale of the spheres. This means that the spheres are 100 times larger than they would be if everything was to scale. To move around the 3D space, we use a cube with an attached camera. The script PlayerController is the one handling the user's inputs. It takes the users Horizontal, Vertical and Height inputs, multiplies them by a speed and the delta time between movements and then adds that to the cube's positions. Also, whenever the Right Mouse button is held, the PlayerController also examines the mouse movement and turns them into the cube's rotation. Finally, we have reached the CustomizationScript, which is only used in the Custom scene. It communicates with the Add Sphere and Start buttons. When clicking on Add Sphere, it activates the Add Planet Window found in the scene. It is filled with Input Fields to fill in the desired values for each variable that corresponds to a sphere. Alongside the Add Planet Window being activated, a sphere instance is created. Every time a text field is modified, the script parses through its content and turns it into a float that is stored in the appropriate variable. Until the input is completed and the user clicks on Ok, these values are kept only in CustomizationScript. Any values that affect the way the sphere is displayed are also processed here. To avoid having the CustomizationScript and the InspectSphere scripts try to to display the same object in different ways, as they do not hold the same values for the object, we have the ready valuable in InspectSphere, of type bool. Whenever a sphere's values are processed by CustomizationScript, ready has a value of false, preventing InspectSphere to wrongly display the sphere. Once the Ok button is clicked, the Add Planet Window is deactivated and the values sent in InspectSphere. After the user clicks on start, the Add Planet and the Start buttons are deactivated and an instance of NW Manager is placed in the scene.

## 4 Program code

Physics.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

/* (C) Mihnea S. Teodorescu & Andrei Dumitriu, November 2019
 */

public class Physics : MonoBehaviour
{
    public struct SphereObject
    {
        public bool exists;
        public double mass, radius;
        public Vector3d pos, vel, acc;
    }
    public SphereObject[] sphere;
    public GameObject[] game_sphere;
    public int num_of_spheres;
    public bool paused = false;
    public int speed = 512;
    public double t = 0, dt = 1;
    private double G = 6.674e-11;
    public GameObject TheText;
    public Text TextField;

    double Magnitude(Vector3d vec)
    {
        return Math.Sqrt(vec.x * vec.x + vec.y * vec.y + vec.z * vec.z);
    }

    Vector3d GravForce(int i, int j)
    {
        Vector3d r_vec, r_hat, force_vec;
        double r_mag, force_mag;
        r_vec = sphere[i].pos - sphere[j].pos;
        r_mag = Magnitude(r_vec);
        r_hat = r_vec / r_mag;
        force_mag = G * sphere[i].mass * sphere[j].mass / (r_mag * r_mag);
        force_vec = -force_mag * r_hat;
        return force_vec;
    }

    void UpdateAcc()
    {
        for (int i = 0; i < num_of_spheres; i++)
        {
            sphere[i].acc = new Vector3d(0, 0, 0);
        }
        for (int i = 0; i < num_of_spheres; i++)
        {
            if (sphere[i].exists)
            {
                for (int j = i+1; j < num_of_spheres; j++)
                {
                    if (sphere[j].exists)
                    {
                        sphere[i].acc += GravForce(i, j) / sphere[i].mass;
                        sphere[j].acc += GravForce(j, i) / sphere[j].mass;
                    }
                }
            }
        }
    }

    void UpdateVelPos()
    {
        for (int i = 0; i < num_of_spheres; i++)
        {
            if (sphere[i].exists)
            {
                sphere[i].vel += sphere[i].acc * dt;
                sphere[i].pos += sphere[i].vel * dt;
            }
        }
    }

    void UpdateGameSpheres()
    {
        for (int i = 0; i < num_of_spheres; i++)
        {
            if (!sphere[i].exists)
            {
                game_sphere[i].SetActive(false);
            }
            game_sphere[i].GetComponent<InspectSphere>().ax = sphere[i].pos.x;
            game_sphere[i].GetComponent<InspectSphere>().ay = sphere[i].pos.y;
            game_sphere[i].GetComponent<InspectSphere>().az = sphere[i].pos.z;
            game_sphere[i].GetComponent<InspectSphere>().rad = sphere[i].radius;
            game_sphere[i].GetComponent<InspectSphere>().vel = sphere[i].vel;
            game_sphere[i].GetComponent<InspectSphere>().acc = sphere[i].acc;
        }
    }

    void MergeCollided()
    {
        for (int i = 0; i < num_of_spheres; i++)
        {
            if (sphere[i].exists)
            {
                for (int j = i+1; j < num_of_spheres; j++)
                {
                    if (sphere[j].exists)
                    {
                        Vector3d r_vec = sphere[i].pos - sphere[j].pos;
                        double r_mag = Magnitude(r_vec);
                        if (r_mag - sphere[i].radius - sphere[j].radius <= 0)
                        {
                            if (sphere[i].mass < sphere[j].mass)
                            {
                                int t = i;
                                i = j;
                                j = t;
                            }
                            sphere[j].exists = false;
                            sphere[i].vel = (sphere[i].vel * sphere[i].mass + sphere[j].vel * sphere[j].mass) / (sphere[i].mass + sphere[j].mass);
                            sphere[i].mass += sphere[j].mass;
                            sphere[i].radius = Math.Pow(Math.Pow(sphere[i].radius, 3) + Math.Pow(sphere[j].radius, 3), 1.0/3.0);
                            UpdateGameSpheres();
                            UpdateAcc();
                        }
                    }
                }
            }
        }
    }

    void Start()
    {
        game_sphere = GameObject.FindGameObjectsWithTag("Sphere");
        num_of_spheres = game_sphere.Length;
        sphere = new SphereObject[num_of_spheres];
        for (int i = 0; i < num_of_spheres; i++)
        {
            sphere[i].exists = true;
            sphere[i].pos = new Vector3d(game_sphere[i].GetComponent<InspectSphere>().ax, game_sphere[i].GetComponent<InspectSphere>().ay, game_sphere[i].GetComponent<InspectSphere>().az);
            sphere[i].vel = game_sphere[i].GetComponent<InspectSphere>().vel;
            sphere[i].mass = game_sphere[i].GetComponent<InspectSphere>().mass;
            sphere[i].radius = game_sphere[i].GetComponent<InspectSphere>().rad;
        }
        TheText = GameObject.FindGameObjectWithTag("Speed Text");
        TextField = TheText.GetComponent<Text>();
        Display();
    }

    public void Display()
    {
        TextField.text = "speed: " + speed.ToString() + " dt: " + dt.ToString();
    }

    void GetInput()
    {
        if (Input.GetKeyDown("space"))
        {
            paused = !paused;
        }
        if (Input.GetKeyDown("escape"))
        {
            SceneManager.LoadScene(0);
        }
        if (Input.GetKeyDown("r") && speed < 262144)
        {
            speed *= 2;
            Display();
        }
        if (Input.GetKeyDown("f") && speed > 1)
        {
            speed /= 2;
            Display();
        }
        if (Input.GetKeyDown("t") && dt < 10000)
        {
            dt *= 10;
            Display();
        }
        if (Input.GetKeyDown("g") && dt > 1)
        {
            dt /= 10;
            Display();
        }
    }

    void Update()
    {
        GetInput();
        if (!paused)
        {
            do
            {
                UpdateAcc();
                UpdateVelPos();
                MergeCollided();
                t += dt;
            }
            while (t % speed != 0);
        }
        UpdateGameSpheres();
    }
}
```

## 5 Featured models

*Newton's World* comes with three pre-created ready-to-test models, as well as with the option to create a custom model of the Universe, capable of featuring any number of spheres. Of course, the user ought to be aware that with each added sphere the necessary computing power increases significantly, given the quadratic time of the program.

- *Earth and Moon*: This model features the Earth along its sole natural satellite, Luna, in an absolutely empty Universe. Because of the reduction of scale described above (distances are reduced by a factor of 100), the two celestial bodies appear to be ridiculously close to each other. However, this is but an illusion and does not affect the simulation in any way.
- *Collision*: This model describes an apocalyptic scenario where the Earth has somehow been deorbited (physically very possible) and is now slowly being accelerated towards the Sun. Eventually the two will collide and the Earth will disappear once and for all in accordance with the *MergeCollided* function.
- *Solar System*: This is a simplified simulation of our very stellar system. It even features Pluto. (Although one might find it rather difficult to actually spot it!) As always, the user is able navigate in three-dimensional space, increase or decrease the run speed of the

simulation, and simply try to get a grasp of how insignificant our tiny planet is in contrast to the ridiculous vastness of the Universe.

- *Custom*: Finally, this mode enables the user to create their own numerical model. Simply click *Add Sphere*, input the data for each sphere, click *Ok*, and run a space simulation.

## 6 Evaluation

Due to our rather simplistic approach, we managed to achieve a functional system which is really easy to build upon for further development. It is worth mentioning that such system is by far not flawless and that there is definitely room for improvement. One issue which is to be dealt with the program's time complexity. This may drastically be improved by splitting the model into several sub-regions. In such an approach, for each generated sub-region only the spheres which belong to its area are to be taken into account and thus getting rid of $\mathcal{O}(n^2)$. We could have gone for such a model, but instead we decided to invest most of our time and energy into the actual study of the mathematics and physics that such a project involves. And as always, we are looking forward to improving our results.

## 7 Credits

- Dinv Studio: background for the Menu
- Pulsar Bytes: the skybox for all the other scenes
- Mark Hutson: $EndlessVoid$ – the background music
- GitHub user sldsmkd: $Math.cs$, $Vector2d.cs$, $Vector3d.cs$