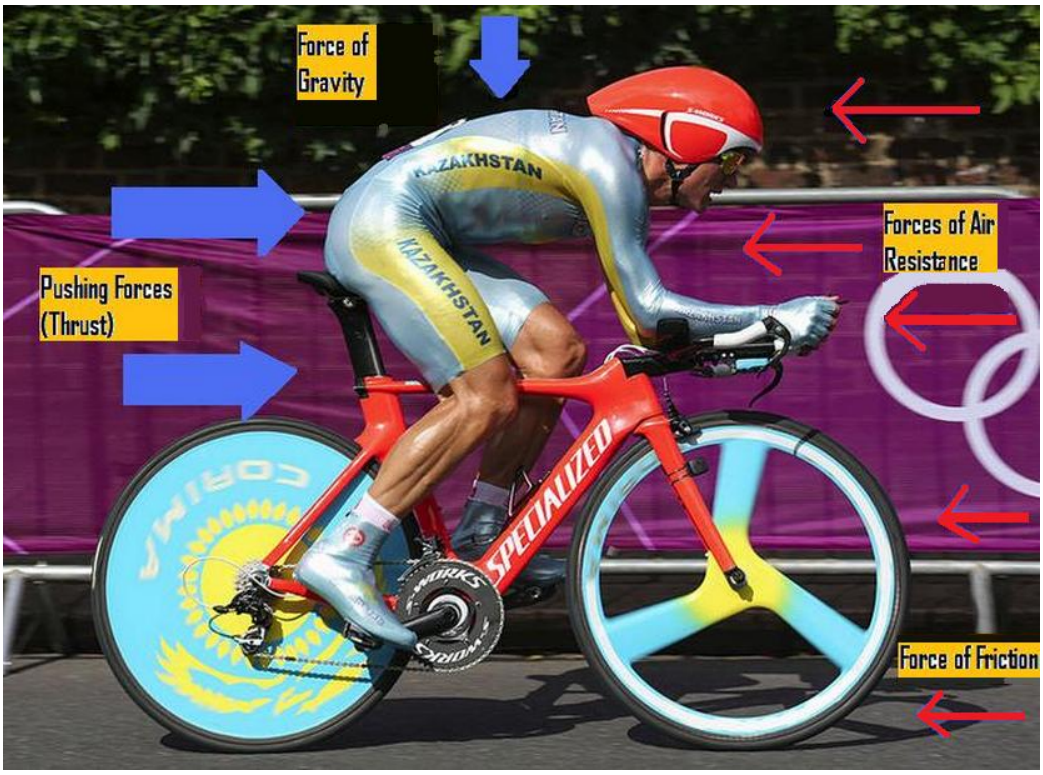
Practical 2: Newtonian Physics

Babis Koniaris



Introduction

The goals of this practical are to:

- Give you an opportunity to become familiar with the Object-Oriented Programming framework (OOP) code that is provided to you.
- Develop a practical understanding of particle simulations

Architecture

The code provided to you this week has evolved using an OOP approach. Whereas last week you focussed on implement some basic particle animations, this week's main challenge is to understand and extend the given framework to run a *dynamic* particle simulation. The OOP approach here allows us to use more intuitive abstractions. The project that you're going to be using is "02_particles_framework". There, you can find the several classes, split into two groups:

The first group are classes that you don't have to change their code at all for your physics simulation (nobody is stopping you however from modifying them, if you wish to):

- **Application**: This class contains the main application logic, is responsible for the window, input, and any other global application state (such as physics and objects)
- **Mesh**: This class represents a 3D mesh (vertices, normals indices and associated GPU buffer information)
- **Shader**: This class represents a GPU shader program (vertex and fragment shaders)
- **Camera**: This class represents a camera in the virtual world

The second group are classes that you need to modify to complete the required tasks:

- **PhysicsEngine**: This class represents the physics state: it stores all physics objects and has code to simulate their movement.

- **PhysicsBody**: This class represents a basic physics object. It stores a pointer to a mesh (the 3D data that make up the object), a pointer to a shader (how should we display the object), a colour (to assist with visualisation), and transformation data: position, scale and orientation. We can use this class to represent immovable objects, such as the ground plane.
- **Particle**: This subclass is derived from a physics object, and adds a few extra variables to track necessary state, such as mass and velocity, that is needed for dynamic simulation.

This framework design is just an example one, and is by no means definitive. After the end of this module, you will possibly have your own ideas on how the framework should be changed to better support whatever physics simulation idea you have in mind!

Application flow

Our main() function is simply the creation of an Application object and running a function called MainLoop() on it. The MainLoop function first initialises the window and input, then initialises the mesh and shader databases, and finally initialises the physics engine, before entering the game loop.

Sidenote: We are using mesh and shader databases to avoid duplication of objects: if every particle needs a non-modifiable reference to a mesh and a shader, we don't need to instantiate unique ones per particle, we can just reuse a single one! For example, instead of creating 100 cubes for each of 100 particles, we can reference the same cube, and use a transformation matrix to place it appropriately in the scene. The transformation matrix is the quantity that is required per-particle, not the actual mesh data, which are shared among particles. Assigning meshes and shaders to particles are the only actions that involve use of pointers in this module, and you will not be required to do any dynamic memory allocation using new/delete/malloc or smart pointers.

In the main loop we do the following:

- Keep track of time
- Handle window/input events in the application

- Handle key events in the PhysicsEngine class (**you can modify this to suit your needs!**)
- Handle input events related to camera movement
- Generate view/projection matrices
- Update the physics engine state
- Draw the objects in the physics engine (this normally would be considered bad design, but is done to keep code simple, compared to introducing more classes and more complex code)

General coding responsibilities

For the tasks this week and all other weeks, there are a few common things that you need to be doing:

- Adding variables in the physics engine class (in the class body, not global variables)
- Initialising variables in the PhysicsEngine::Init function
- Updating the physics state in the PhysicsEngine::Update function
- For interactive experiments, handle keys in PhysicsEngine::HandleInputKey
- To display any new objects, add relevant code in PhysicsEngine::Display (see example code)

Of course at various points you might need to introduce new member variables and functions in the PhysicsEngine class (or even in Particle/PhysicsBody classes) if you see fit to do so. The framework is designed to guide you, but not be overly restrictive.

Dynamic simulation

This section is very important, as it describes the sequence of actions that makes a dynamic particle simulation. It is based on two key concepts that will be discussed in more detail in the next lecture, but that this summary should allow you to implement: Newton's second law and numerical integration

Newton's second law

Most of you will be familiar with the famous equation:

$$\boldsymbol{F} = m\boldsymbol{a} \tag{1}$$

This is all we need to carry out particle simulations: a link between forces applied to a particle and its acceleration. For the time being, the only force that we know applies to the particles, that we need to simulate, is gravity. In the uniform gravity field of earth, we have:

$$\boldsymbol{F}_g = m\boldsymbol{g} \tag{2}$$

where:

$$\boldsymbol{g} = (0, -9.81, 0)\ ms^{-2} \tag{3}$$

If we knew of other forces, for example aerodynamic drag ($\boldsymbol{F}_a$) or friction ($\boldsymbol{F}_f$), we would just add them up:

$$\boldsymbol{F} = \boldsymbol{F}_g + \boldsymbol{F}_a + \boldsymbol{F}_f \tag{4}$$

Numerical integration

Once we know the acceleration of a particle, we can obtain its velocity and position by integration. Remember, $\boldsymbol{v}$ is the derivative of $\boldsymbol{r}$ ($\boldsymbol{v} = d\boldsymbol{r}/dt$) and $\boldsymbol{a}$ is the derivative of $\boldsymbol{v}$ ($\boldsymbol{a} = d\boldsymbol{v}/dt$). So we get $\boldsymbol{v}$ from $\boldsymbol{a}$ by integration and $\boldsymbol{r}$ from $\boldsymbol{v}$ by integration. Next week, you will learn about several numerical integration techniques. Until then, here's one you can use that will work perfectly well for the type of simulations you are implementing.

Let's call $\boldsymbol{v}_{n+1}$ the new velocity to be computed at each step and $\boldsymbol{v}_n$ the velocity at the previous iteration. If we adopt the same notations for the acceleration and position, we can operate our integration as follows:

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + h\boldsymbol{a}_n \tag{5}$$
$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + h\boldsymbol{v}_{n+1} \tag{6}$$

where h represents the duration of a time step (*deltaTime* in the code)

Simulation process

Now that you know about Newton's second law and a numerical integration method, you can implement a simulation as follows:

For each frame and particle:

- Add all forces applied to a particle: $\boldsymbol{F}$
- $\boldsymbol{F} = m\boldsymbol{a}$ gives you the acceleration
- Using an integrator (such as the one above), compute the new velocity and position of the particle.

This should be sufficient for you to complete this week's tasks.

Tasks

This week's tasks will echo last week's in that the outcome will look similar, but the algorithms will be very different. Instead of precomputed animations and closed-form functions, you will implement genuine realtime simulations. The focus of this week is to get things to work within the given framework.

Task 1: Particle simulation

You should simulate a particle, subject only to gravity. The result should look very similar to what you did last week. For testing, try giving your particle various initial velocities, and make sure that the simulation looks like what you would expect.

Task 2: Particle simulation, with simple collision detection

You should add a simple collision detection step to allow particles to bounce on the ground plane. To make it more interesting than last week, this time you are asked to implement collisions with the 6 walls of a cube in which the ball bounces. To help you implement this collision detection algorithm, you can represent the cube as two vectors:

- **center**. The center of the cube
- **size**. The size of the cube (width, height, depth)

In addition to making a particle bounce around in a cubic room, amend your collision response so that energy is absorbed at every bounce. The particle should eventually come to a standstill.

You can try using the provided cube model, whose mesh is centered at $(0, 0, 0)$ and has a size of 2 (so it ranges from $(-1, -1, -1)$ to $(1, 1, 1)$).

Task 3: Aerodynamic drag

As you know, in the real world (within the earth's atmosphere), a small object is subject to more than gravity. Aerodynamic drag in particular has a considerable impact on the motion of objects traveling through a fluid (like air), the higher the velocity, the stronger the effect.

You should add aerodynamic drag to the forces applied to simulated particles.

Further tasks

Here are further tasks you can complete to further your understanding of the topic:

- Add wind to your simulation. You can start with a simple isotropic wind that has constant velocity and take things further by varying the wind speed and then making it anisotropic i.e., not the same across the 3D space.
- Add friction to your simulation and/or something that will ensure that your particles don't keep travelling forever. In other words, make them stop in a physically believable fashion.