Computer Seminar I: Final Project
Simulation of Charged Particle Inside Electromagnetic Field

Farros Alferro

May 9^{th}, 2022

1 Introduction

This simulation is based on charged particles' natural behavior when it interacts with other charged particle, a static magnetic field, and a static electric field. It will visualize the trajectory of either a positive or negative particles governed by the existing law such as *Lorentz force* and *Coulomb's law*. However, in this simulation it is assumed that there is no friction (vacuum) and the gravity acceleration is set to 0.

2 Program Explanation

When the program is executed, it will show a pygame window with dimension 1200×800 pixels and blue-colored background. It has two legends at the right top part that indicate the vector of magnetic and electric field respectively ($[x, y, z]$). There is also a short text instructing on how to start over by pressing the $[r]$ key button located at the bottom left part of the window.

The detailed commands are described as follows:

- Left-Click Mouse: Create a positive charge particle that is indicated by + sign and red color.
- Right-Click Mouse: Create a negative charge particle that is indicated by − sign and elm color.
- Key $[SPACE]$ Button: Has to be pressed together with either left or right click mouse to create a white-colored fixed charged particle. The charge depends on the mouse click.
- Key $[UP]$ and $[DOWN]$ Button: Change the value of the electric field vector in y-direction.
- Key $[LEFT]$ and $[RIGHT]$ Button: Change the value of the electric field vector in x-direction.
- Key $[EQUAL]$ and $[-]$ Button: Change the value of the magnetic field vector in z-direction.
- Key $[r]$ Button: Delete all the particles and reset both fields vectors to 0.

In addition, the initial velocity of the charged particles can be set by clicking the mouse and dragging it a distance before releasing it (similar to the exercise given in the 5^{th} lecture). There is also a footprint for each non-fixed particle when it moves.

3 Implementation

In general, most of the implementations that are being used in this simulation are based on the *spring_mass.py* and *interactive_main.py* files. The program also has two .py files that consists of *electro_magnetic.py*, which holds the classes of the objects that are going to be implemented, and *main.py* that sets the initial values as well as runs the program. Moreover, the *pygame.math.Vector3* module is used here instead of the *pygame.math.Vector2* as 3D vectors are needed to implement the *Lorentz force*.

3.1 World, CircleDrawer, Tracer, and ValueBoard Classes

3.1.1 World Class

```python
class World:
    def __init__(self, size, dt):
        self.size = size
        self.dt = dt

```

Figure 1: World Class

The World Class is pretty much the same as the World Class defined in *spring_mass.py* file except it doesn't have the gravity argument as this simulation doesn't take gravity force into account.

3.1.2 CircleDrawer Class

```python
class CircleDrawer:
    def __init__(self, color, text_color, font_size,
                 font_file, width, antialias=True):
        self.color = pygame.Color(color)
        self.color_text = text_color
        self.width = width
        self.font_size = font_size
        self.font = pygame.font.Font(font_file, self.font_size)
        self.antialias = antialias

    def __call__(self, screen, center, radius, text):
        pygame.draw.circle(screen, self.color, center, radius, self.width)
        text_image = self.font.render(text, self.antialias, self.color_text)
        screen.blit(text_image,
                    (center - PgVector((0.5 * radius, (self.font_size - 2 * radius)))
```

Figure 2: CircleDrawer Class

The CircleDrawer Class used in this simulation was also based on the CircleDrawer Class from *spring_mass.py* file. This class will be used only to draw the charged particles and since there are two types of it, it will be more convenient to add a text indicating what kind of charge it is. This feature has already included in this Class indicated by the addition of several arguments like `font_size` and `font_file` on the initiation to specify the font and `text` on the `__call__` method to print the desired text. The position of the text (in this case is sign) is in the center of the circle and was determined by trial and error with `radius` = 10 and `font_size` = 30. However, the position of the text has to be adjusted once the mentioned values are changed.

3.1.3 Tracer Class

```python
class Tracer:
    def __init__(self, color, radius, interval, width):
        self.color = pygame.Color(color)
        self.width = width
        self.interval = interval
        self.radius = radius

    def __call__(self, screen, position_list):
        i = 0
        for i in range(0, (len(position_list) -
                                len(position_list)%self.interval), self.interval):
            pos = position_list[i]
            pygame.draw.circle(screen, self.color, (pos[0], pos[1]), self.radius)
```

Figure 3: Tracer Class

This is a new class functioned as the tracer for the moving particles. When it is called, it receives an argument `position_list`, which is a list that contains the position of the particle for every time interval. The tracer then will be displayed according to the specified interval. In addition, since it traces the particle based on its previous positions, it will indicate the particles' acceleration (the same concept as ticker tape).

3.1.4 ValueBoard Class

```python
class ValueBoard:
    def __init__(self, color, font_size, font_file, pos, antialias=True):
        self.color = pygame.Color(color)
        self.font_size = font_size
        self.font = pygame.font.Font(font_file, font_size)
        self.pos1 = PgVector((pos[0], pos[1]))
        self.pos2 = self.pos1 + 0.8 * PgVector((0, self.font_size))
        self.antialias = antialias

    def __call__(self, screen, text1, text2):
        text_image1 = self.font.render(text1, self.antialias, self.color)
        text_image2 = self.font.render(text2, self.antialias, self.color)
        screen.blit(text_image1, self.pos1)
        screen.blit(text_image2, self.pos2)
```

Figure 4: ValueBoard Class

The main purpose of this class is to display the legend text. It receives several text-related input as initiation arguments and two text input when it is called. The second text (`text2`) will be displayed below the first text (`text1`).

3.2 Creating Particles

3.2.1 Non-Fixed Charged Particles

```python
class ChargedParticle:
    def __init__(self, pos, vel, world, radius=10, mass=10,
                 charge_value=10, charge_sign=True, restitution=0.95,
                 drawer=None, tracer=None):

        self.is_alive = True
        self.world = world
        self.drawer = drawer
        self.tracer = tracer

        self.pos = Vector3d((pos[0], pos[1], 0))
        self.list_position = [(pos[0], pos[1], 0)]
        self.vel = Vector3d((vel[0], vel[1], 0))
        self.radius = radius
        self.mass = mass
        self.restitution = restitution
        if charge_sign:
            self.charge = charge_value
            self.text = '+'
        else:
            self.charge = -charge_value
            self.text = '-'

        self.total_force = Vector3d((0, 0, 0))
```

Figure 5: Non-Fixed Charged Particles Initiation

Overall, the initiation arguments are pretty much the same as `PointMass` class. There are some additional arguments like `charge_value`, `charge_sign`, and `tracer` (the name reflects the role).

The methods used are also pretty similar to `PointMass` class as shown in figure 6. There are some new methods like `trace` to call the object of type `Tracer` and `update_after_move` to kill the particle once it moves beyond the world. The same function is also used to calculate the new position and velocity (`integrate_sympletic`, figure 7).

```python
def integrate_symplectic(pos, vel, force, mass, dt):
    vel_new = vel + force / mass * dt
    pos_new = pos + vel_new * dt
    return pos_new, vel_new
```

Figure 7: Integrate_Sympletic Function

```python
    def update(self):
        self.move()
        self.list_position.append(self.pos)
        self.update_after_move()
        self.total_force = Vector3d((0, 0, 0))

    def draw(self, screen):
        self.drawer(screen, PgVector((self.pos[0], self.pos[1])),
                    self.radius, self.text)

    def trace(self, screen):
        self.tracer(screen, self.list_position)

    def receive_force(self, force):
        self.total_force += Vector3d(force)

    def update_after_move(self):
        if self.pos[0] < 0 or self.pos[0] > self.world.size[0] \
            or self.pos[1] > self.world.size[1] or self.pos[1] < 0:
            self.is_alive = False

    def move(self):
        self.pos, self.vel = \
            integrate_symplectic(self.pos, self.vel, self.total_force,
                                 self.mass, self.world.dt)
```

Figure 6: Non-Fixed Charged Particles Methods

3.2.2 Fixed Charged Particles

This class also uses the same implementation as `FixedPointMass`: inheritance (figure 8). The charge sign is also set as the initiation argument as it will depend on the mouse-click input. By setting the velocity value to $(0, 0, 0)$ and mass to 10^9, the particle won't move anywhere.

3.3 Forces

There are 4 forces in total: *Lorentz force*, Electric field force, *Coulomb force*, and Collision force. The first two forces are not appended to the actor list as its values are subject to change, whereas the latter are appended to the actor list because it governs interaction between particles thus needs to track the existing particle. Each force has its own function (the physical equation) and class (the regulator).

3.3.1 *Lorentz Force*

Lorentz force dictates that a particle q moving with a velocity $\vec{v}$ will experience a force of

$$\vec{F} = q\vec{v} \times \vec{B}$$

due to the existence of a magnetic field $\vec{B}$. This equation is implemented in `compute_lorentz_force` function as shown in figure 9.

The class that corresponds to this function is shown in figure 10.

```python
class FixedChargedParticle(ChargedParticle):
    def __init__(self, pos, vel, world, radius=10, mass=10,
                 charge_value=10, charge_sign=True, restitution=0.95,
                 drawer=None, tracer=None):
        super().__init__(pos, vel, world, radius, mass,
                         charge_value, charge_sign, restitution, drawer, tracer)
        self.vel, self.mass = Vector3d((0, 0, 0)), 1e9

    def move(self):
        pass
```

Figure 8: Fixed Charged Particles

```python
def compute_lorentz_force(magnetic_field, vel, charge):
    return vel.cross(charge * magnetic_field)
```

Figure 9: Lorentz Force Function

```python
class MagneticForce:
    def __init__(self, world, actor_list, MagneticField=(0,0,0),
                 target_condition=None, drawer=None, tracer=None):
        self.world = world
        self.actor_list = actor_list
        self.magnetic_field = Vector3d((MagneticField[0], MagneticField[1],
                                        MagneticField[2]))
        self.drawer = drawer
        self.tracer = tracer

        if target_condition is None:
            self.target_condition = is_charged_particle
        else:
            self.target_condition = target_condition

    def update(self):
        self.generate_force()

    def draw(self, screen):
        text1 = "Magnetic Field Vector:"
        text2 = str(self.magnetic_field)
        self.drawer(screen, text1, text2)

    def trace(self, screen):
        if self.tracer is not None:
            self.tracer(screen)

    def generate_force(self):
        plist = [a for a in self.actor_list if self.target_condition(a)]
        for p in plist:
            fl = compute_lorentz_force(self.magnetic_field, p.vel, p.charge)
            p.receive_force(fl)
```

Figure 10: Lorentz Force Class

The initiation includes the declaration of several variables like `self.magnetic_field`, `self.drawer`, `self.target_condition`, etc. The overall structure is the same as the `CollisionResolver` Class: the `generate_force` method will iterate over the objects inside the actor list that fulfill the `self.target_condition` (which later will be defined as function that excludes anything except particles, as shown in figure 11), then it will compute the corresponding *Lorentz force* for each particle. This method will then be executed once the `update` method is called.

```python
def is_charged_particle(actor):
    return isinstance(actor, ChargedParticle)
```

Figure 11: is_charge_particle Function

For the `draw` method, since we want to display texts that indicate the value of the magnetic field vector, we create the indicator text "Magnetic Field Vector:" and the value of the magnetic field itself `self.magnetic_field`. Later in the *main.py*, we will declare the `self.drawer` as the `ValueBoard` Class. Moreover, the *Lorentz* force does not need any tracer thus `self.tracer` is defined as shown in line 255-257.

3.3.2 Electric Field Force

This law dictates that a particle q will experience a force of

$$\vec{F} = q\vec{E}$$

due to the existence of a electric field $\vec{E}$. This equation is implemented in `compute_electric_field_force` as shown in figure 12

```python
def compute_electric_field_force(electric_field, charge):
    return electric_field * charge
```

Figure 12: Electric Field Force Function

The class that corresponds to this function is shown in figure 13.

In general, since this force behaves in the same way as *Lorentz force*, it is implemented with the same structure as `MagneticForce` Class.

3.3.3 *Coulomb Force*

Coulomb's law dictates that there exists a force between two electrically charged particles:

$$\vec{F} = k_e \frac{q_1 1_2}{|\mathbf{r}_{12}|^2} \hat{\mathbf{r}}_{12}$$

Where k_e is *Coulomb's constant*, q_1 and q_2 are the charges, $\mathbf{r}_{12}$ is the vectorial distance between the charges and $\hat{\mathbf{r}}_{12}$ is the unit vector pointing from q_2 to q_1. This force is implemented in `coulomb_force` as shown in figure 14.

```python
class ElectricFieldForce:
    def __init__(self, world, actor_list, ElectricField=(0,0,0),
                 target_condition=None, drawer=None, tracer=None):
        self.world = world
        self.actor_list = actor_list
        self.electric_field = Vector3d(ElectricField[0], ElectricField[1],
                                       ElectricField[2])
        self.drawer = drawer
        self.tracer = tracer

        if target_condition is None:
            self.target_condition = is_charged_particle
        else:
            self.target_condition = target_condition

    def update(self):
        self.generate_force()

    def draw(self, screen):
        text1 = "Electric Field Vector:"
        text2 = str(self.electric_field)
        self.drawer(screen, text1, text2)

    def trace(self, screen):
        if self.tracer is not None:
            self.tracer(screen)

    def generate_force(self):
        plist = [a for a in self.actor_list if self.target_condition(a)]
        for p in plist:
            fe = compute_electric_field_force(self.electric_field, p.charge)
            p.receive_force(fe)
```

Figure 13: Electric Field Force Class

```python
132 def coulomb_force(constant, p1, p2):
133     if p1.pos == p2.pos:
134         return None
135     direction = p2.pos - p1.pos
136     distance = (direction.magnitude())
137     unit_vector = direction / distance
138     effective_distance = distance - (p1.radius + p2.radius)
139     fe = unit_vector * ((constant * p1.charge * p2.charge /
140                          (effective_distance ** 2)))
141     return fe
```

Figure 14: Coulomb Force Function

The condition in line 133 is written because the direction of the force cannot be determined if the positions of particles 1 and 2 are exactly the same. This is actually an improbable situation that might happen in the simulations. Here, it is ignored. In addition, I also tweaked the formula a little bit: instead of using the actual distance between particles, I used the variable `effective_distance`, which is defined as the distance subtracted by both particles' radius, as the denominator.

The reason why I did this is because I just found out that charged particles (e.g. electron and proton) cannot in normal situations 'collide' because of 'strong nuclear force'. Even though I have revised the collision impact force a little so that it would collide with a certain distance between (it can be seen as the effect of the strong nuclear force), there are still some cases where the *Coulomb force* is higher than the impact force resulting one particle to penetrate the other. In my opinion, it is more natural for the particles to just fly away with infinite force before the penetration happens.

The class that corresponds to this function is shown in figure 15

```
158 class CoulombForce:
159     def __init__(self, world, actor_list, constant, target_condition=None,
160                  drawer=None, tracer=None):
161         self.is_alive = True
162         self.world = world
163         self.actor_list = actor_list
164         self.drawer = drawer
165         self.constant = constant
166         self.tracer = tracer
167 
168         if target_condition is None:
169             self.target_condition = is_charged_particle
170         else:
171             self.target_condition = target_condition
172 
173     def update(self):
174         self.generate_force()
175 
176     def draw(self, screen):
177         if self.drawer is not None:
178             self.drawer(screen)
179 
180     def trace(self, screen):
181         if self.tracer is not None:
182             self.tracer(screen)
183 
184     def generate_force(self):
185         plist = [a for a in self.actor_list if self.target_condition(a)]
186         n = len(plist)
187         for i in range(n):
188             for j in range(i+1, n):
189                 p1, p2 = plist[i], plist[j]
190                 fe = coulomb_force(self.constant, p1, p2)
191                 if fe is None:
192                     continue
193                 p2.receive_force(fe)
194                 p1.receive_force(-fe)
```

Figure 15: *Coulomb* Force Class

This class also has the same structure as the previous classes. It does not need any legend thus the drawer method (line 176) is written in a similar way as the trace method. As we want the *Coulomb force* to exists between all generated particles, we do the iteration two times: (a) over all the particles, (b) from the current iterated particle plus one until the end of the list (as shown in line 187-188). It has the same idea as the `generate_force` method in the `CollisionResolver` class.

3.3.4 Collision Force

It has the same code as the `compute_impact_force_between_points` function as shown in figure 16.

```
def compute_impact_force_between_points(p1, p2, dt, feynman_radius, constant):
    if (p1.pos - p2.pos).magnitude() > p1.radius + p2.radius + feynman_radius:
        return None
    if p1.pos == p2.pos:
        return None
    normal = (p2.pos - p1.pos).normalize()
    v1 = p1.vel.dot(normal)
    v2 = p2.vel.dot(normal)
    if v1 < v2:
        return None
    e = p1.restitution * p2.restitution
    m1, m2 = p1.mass, p2.mass
    f1 = constant * normal * (-(e + 1) * v1 + (e + 1) * v2) / (1/m1 + 1/m2) / dt
    return f1
```

Figure 16: Collision Force Function

However, as I have mentioned before, I tweak it a little so that the particles will collide at a certain distance. That certain distance is declared as an input variable called `feynman_radius`. Moreover, there is another additional argument called `constant` which is used to amplify the impact force.

The reason why I use collision force instead of putting the `feynman_radius` in the *Coulomb force* denominator (so the effective distance will be the distance between particles subtracted by both particles' radius and `feynman_radius`) is because if I took the latter approach, the particles will move in high speed due to the infinite force when it reaches the specified distance. I have no idea of how does the strong nuclear force governs the particles' interaction, but I think it is better to just use the collision force as it seems more natural.

The class that corresponds to this function is shown in figure 17

```python
class CollisionResolver:
    def __init__(self, world, actor_list, feynman_radius, constant,
                 target_condition=None, drawer=None, tracer=None):
        self.is_alive = True
        self.world = world
        self.drawer = drawer
        self.tracer = tracer
        self.feynman_radius = feynman_radius
        self.constant = constant

        self.actor_list = actor_list
        if target_condition is None:
            self.target_condition = is_charged_particle
        else:
            self.target_condition = target_condition

    def update(self):
        self.generate_force()

    def draw(self, screen):
        if self.drawer is not None:
            self.drawer(screen)

    def trace(self, screen):
        if self.tracer is not None:
            self.tracer(screen)

    def generate_force(self):
        plist = [a for a in self.actor_list if self.target_condition(a)]
        n = len(plist)
        for i in range(n):
            for j in range(i + 1, n):
                p1, p2 = plist[i], plist[j]
                f1 = compute_impact_force_between_points(p1, p2, self.world.dt,
                                                         self.feynman_radius,
                                                         self.constant)
                if f1 is None:
                    continue
                p1.receive_force(f1)
                p2.receive_force(-f1)
```

Figure 17: Collision Force Class

It has the same code as the original except that it accepts additional argumen such as `feynman_radius` and `constant` as it will be used on the `generate_force` method.

3.4 *main.py*

The code contained in this file has the same implementation as the code in *interactive_main.py* from lecture 7 which has two main class: `ActorFactory` and `AppMain` class.

3.4.1 `ActorFactory` Class

This is the class where all the variables that won't be changed throughout simulation are declared. Values like mass, radius, font size, color, *Coulomb* constant, and others are declared. Then each method will accordingly return an object of type of class that exists in the *electro_magnetic.py* file with some of the initiation arguments are filled with the declared variables. The class is shown in figure 18 and 19.

```
5  class ActorFactory:
6      def __init__(self, world, actor_list):
7          self.world = world
8          self.actor_list = actor_list
9  
10     def create_charged_particle(self, pos, x0, y0, charge_sign=True, fixed=False):
11         x, y = pos
12         vel = ((x0 - x)/10, (y0 - y)/10, 0)
13         mass = 10
14         radius = 10
15         tracer_radius = 3
16         restitution = 0.95
17         charge_value = 0.3
18         tracer_interval = 5
19         charge_sign_color = (29, 53, 87)
20         font_size = 30
21         font_file = None
22 
23         if not fixed:
24             ChargedParticleClass = emf.ChargedParticle
25             if charge_sign:
26                 color = (211, 59, 82)
27             else:
28                 color = (29, 115, 139)
29         else:
30             ChargedParticleClass = emf.FixedChargedParticle
31             color = (255, 255, 255)
32 
33         return ChargedParticleClass(pos, vel, self.world, radius, mass, charge_value,
34                                     charge_sign, restitution,
35                                     emf.CircleDrawer(color, charge_sign_color, font_size,
36                                                      font_file, width=0),
37                                     emf.Tracer(color, tracer_radius, tracer_interval,
38                                                width=0))
39 
```

Figure 18: `ActorFactory` Class (1)

In the `create_charged_particle` method, I add a condition to specify the color that will be assigned to the created particle. If it is not a fixed particle, it will have red color for positive charge and elm color for negative charge. If it is a fixed particle, then it will display white color (line 23-31). Moreover, the velocity is calculated by taking the difference between the initial position and the final position, which later will be specified when mouse button is down and up respectively.

```python
    def generate_coulomb_force(self):
        coulomb_constant = 500000
        return emf.CoulombForce(self.world, self.actor_list, coulomb_constant)
    
    def generate_magnetic_force(self):
        initial_magnetic_field = Vector3d((0,0,0))
        color = (9, 31, 38)
        font_size = 30
        font_file = None
        pos = (self.world.size[0] - 225, 0)
        return emf.MagneticForce(self.world, self.actor_list, initial_magnetic_field,
                                 drawer=emf.ValueBoard(color, font_size, font_file,
                                                       pos, antialias=True))
    
    def generate_electric_field_force(self):
        initial_electric_field = Vector3d((0,0,0))
        color = (9, 31, 38)
        font_size = 30
        font_file = None
        pos = (self.world.size[0] - 225, 60)
        return emf.ElectricFieldForce(self.world, self.actor_list,
                                      initial_electric_field, drawer=emf.ValueBoard(
                                          color, font_size, font_file, pos,
                                          antialias=True))
    
    def create_collision_resolver(self):
        feynman_radius = 15
        weak_force_limit = 5
        return emf.CollisionResolver(self.world, self.actor_list,
                                     feynman_radius, weak_force_limit)
```

Figure 19: `ActorFactory` Class (2)

3.4.2 `AppMain` Class

This class is the class where particles and forces will be generated, updated, and displayed. Most of the contents are the same as `AppMain` class in *interactive_main.py*, but some new methods and refinements were added to make the simulation more interactive. This class is shown in figure 20, 21, and 22

```python
class AppMain:
    def __init__(self):
        pygame.init()
        width, height = 1200, 800
        self.screen = pygame.display.set_mode((width, height))
        self.actor_list = []
        self.world = emf.World((width, height), dt=1.0)
        self.factory = ActorFactory(self.world, self.actor_list)
        self.magnetic_force = self.factory.generate_magnetic_force()
        self.electric_field_force = self.factory.generate_electric_field_force()
        self.text_display = self.factory.text_display()

        self.actor_list.append(self.factory.create_collision_resolver())
        self.actor_list.append(self.factory.generate_coulomb_force())

    def add_particle(self, pos, x0, y0, button):
        if pygame.key.get_pressed()[pygame.K_SPACE]:
            fixed = True
        else:
            fixed = False

        if button == 1:
            charge_sign = True
        elif button == 3:
            charge_sign = False
        else:
            return

        p = self.factory.create_charged_particle(pos, x0, y0, charge_sign, fixed)
        self.actor_list.append(p)

    def changing_magfield_value(self, button):
        if button == pygame.K_EQUALS:
            self.magnetic_force.magnetic_field += Vector3d((0,0,1))
        elif button ==  pygame.K_MINUS:
            self.magnetic_force.magnetic_field -= Vector3d((0,0,1))
        else:
            return
```

Figure 20: `AppMain` Class (1)

The `__init__` method is pretty obvious: it creates the actor list and assign each variable to its corresponding object via `ActorFactory` class. It also appends the object of type `CollisionResolver` and object of type `CoulombForce` to the actor list. On the other hand there are some conditions similar to `create_charged_particle` in the `add_particle` method, but it specifies the sign instead of color. In addition, there is also a condition to increase or decrease the magnetic field vector inside the `changing_magfield_value` method.

```python
    def changing_elfield_value(self, button):
        if button == pygame.K_UP:
            self.electric_field_force.electric_field += Vector3d((0,2,0))
        elif button == pygame.K_DOWN:
            self.electric_field_force.electric_field -= Vector3d((0,2,0))
        elif button == pygame.K_RIGHT:
            self.electric_field_force.electric_field += Vector3d((2,0,0))
        elif button == pygame.K_LEFT:
            self.electric_field_force.electric_field -= Vector3d((2,0,0))
        else:
            return

    def reset(self, button):
        if button == pygame.K_r:
            self.actor_list[:] = []
            self.actor_list.append(self.factory.create_collision_resolver())
            self.actor_list.append(self.factory.generate_coulomb_force())
            self.magnetic_force.magnetic_field = Vector3d((0,0,0))
            self.electric_field_force.electric_field = Vector3d((0,0,0))

    def update(self):
        self.magnetic_force.update()
        self.electric_field_force.update()
        for a in self.actor_list:
            a.update()
        self.actor_list[:] = [a for a in self.actor_list if a.is_alive]

    def draw(self):
        self.screen.fill(pygame.Color(208, 224, 239))
        self.electric_field_force.draw(self.screen)
        self.magnetic_force.draw(self.screen)
        self.text_display(self.screen, 'Press [r] to reset', '')
        for a in self.actor_list:
            a.trace(self.screen)
            a.draw(self.screen)
        pygame.display.update()
```

Figure 21: `AppMain` Class (2)

The `changing_elfield_value` method behaves exactly the same as `changing_magfield_value` method, it changes the electric field vector when particular keys are being pressed. For the `reset` method, it will empty the actor list then add again the `CollisionResolver` and `CoulombForce` classes. It will also set the magnetic and electric field equal to zero. The `update` and `draw` method function the same as the original code: it updates the force value and display the particles as well as the legends respectively.

```python
    def run(self):
        clock = pygame.time.Clock()
        
        while True:
            frames_per_second = 60
            clock.tick(frames_per_second)
            
            should_quit = False
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    should_quit = True
                elif event.type == pygame.KEYDOWN:
                    self.changing_magfield_value(event.key)
                    self.changing_elfield_value(event.key)
                    self.reset(event.key)
                    if event.key == pygame.K_ESCAPE:
                        should_quit = True
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    x0, y0 = event.pos
                elif event.type == pygame.MOUSEBUTTONUP:
                    self.add_particle(event.pos, x0, y0, event.button)
            
            if should_quit:
                break

            self.update()
            self.draw()
            
        pygame.quit()
        
if __name__ == "__main__":
    AppMain().run()
```

Figure 22: `AppMain` Class (3)

Lastly, the **run** method will run the program as the original code did.

4 Remark and Possible Improvisation

This section contains the defects and further improvisations that can be made on this program.

1. Several predetermined values like mass, charge, electric field, and others can be set following a number inputted by the user. By specifying these variables' values, the user can learn how might one value influence the overall motion. The values that I set in the previous figures were estimated by trial and error, then the numbers combination that represents the big picture of the particles' behavior (by matching it with pictures from books and internet) was chosen.

2. Under the influence of constant magnetic field, charged particles should move in a circle trajectory with constant radius. However, in this simulation, that does not occur. The radius tends to get bigger and the velocity also becomes higher. One plausible reason for this phenomena is because we make the equation of motion of the particle follows discrete-time basis rather than continuous. Even though the time interval is very small, it affects the *Lorentz force* as this force changes only the direction of the velocity, not the value (which I think is pretty sensitive in discrete basis).

3. The most annoying part of this simulation is when particles with different charge interact. It is most likely that they will move toward each other, resulting in collision at `feynman_radius`. There are some cases where the impact force is not large enough to overcome the *Coulomb* force, resulting the particles to collide, penetrating the `feynman_radius` (or even penetrating toward each other). Moreover, there are also cases where the particle will be launched in a direction parallel to its initial velocity direction where it is supposed to be bounced and move in the opposite direction. This might occur as the *Coulomb* force will be set to infinity once the particles touch each other. This 'infinity' does not specify direction, resulting the mentioned problem. I think the most simple yet difficult approach to solve this issue is by adding some correction terms to the *Coulomb* force, so that the potential will fall (attracting force) and then rise significantly (turns to repulsive force) once the distance between particles reach a certain value. It is simple because once we know the equation, we only have to tweak the corresponding function. But it is hard as I don't have enough knowledge in this field.

4. About the dragging and clicking to set the initial velocity, if user clicks the left mouse button and drag it, then user also clicks the right mouse button, and then drag it again, the initial velocity of positive charge (produced by left click) will be set relative to the position where the right click was pressed, not relative to the position where the left click was pressed. This happens as the initial relative position x_0 and y_0 are set whenever the mouse is clicked (only button 1 and 3 are taken into account). Thus, when user clicks the right mouse button it will initiate new relative position. Furthermore, it is better to give some sort of dashed line that indicates the distance travelled by the cursor relative to its initial clicked position. I have tried to add this feature with several approaches but it was not displayed. I think I made some mistakes on the position or function declaration of the extra code.

5. Once the aforementioned problems are fixed, I think it is better to update the program so that it will visualize particles' movement in 3 dimension. In 3 dimension, the magnetic field vector is not constrained to only z-direction. Many more possibilities can occur in 3 dimension.

6. It is also possible to create a maze-like game where user has to send a particle to a predetermined point in which several fixed particles have been positioned to disturb the motion of the moving particle. User has to choose the suitable particle and set its initial velocity by clicking and dragging the mouse. In my opinion, it will be a decent game to simulates one's idea about charged particles' movement inside electromagnetic field.

5 Closing

This simulation was developed using the concepts taught in the class. I experienced the benefits and drawbacks of using object-oriented program (OOP) approach. It was difficult to fully understand the concept the first time I read about this, but thanks to the final project, I can comprehend the big picture of this concept.

The journey of Computer Seminar I was fun and full of challenges. I learned a lot of new syntax and functions in python, and I was able to grasp it even better thanks to the given exercises. This course has helped me a lot on understanding and implementing python language and I believe it will aid me even more in the future as I'm thinking on becoming a data scientist. Lastly, I want to express my gratitude to all TAs and Professors for all the supports that had been given to me and other IMAC-U students. Thank you!