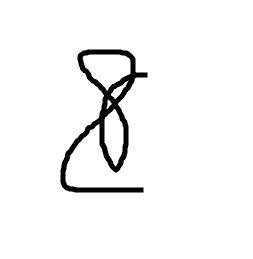
**КУРСОВОЙ ПРОЕКТ**

**Моделирование динамической сцены, вариант № 1**

по дисциплине «Компьютерная графика»

****

|  |  |
| --- | --- |
| Выполнил  студент гр. 5130904/10101 | Абраамян А. М. |
| Преподаватель | Леонтьева Т.В. |

ЗАДАНИЕ

НА ВЫПОЛНЕНИЕ КУРСОВОЙ РАБОТЫ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| студенту группы | | *5130904/10101* | |  | *Абраамяну Александру Манвеловичу* | | | | | | | | |
|  | | *(номер группы)* | |  | *(фамилия, имя, отчество)* | | | | | | | | |
| ***1. Тема проекта (работы)*** | | | | *Моделирование динамической сцены,* | | | | | | | | | |
| *вариант 8* | | | | | | | | | | | | | |
| ***2. Срок сдачи студентом законченного проекта (работы)*** | | | | | | | | | | | | | *17.12.2024* |
| ***3. Исходные данные к проекту (работе)*** | | | | | | | | | *Задание с параметрами* | | | | |
| *согласно варианту, среда разработки PyCharm, графическая* | | | | | | | | | | | | | |
| *библиотека OpenGL* | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | |
| ***4.Содержание пояснительной записки***(перечень подлежащих разработке вопросов): | | | | | | | | | | | | | |
| *Задание, описание алгоритма, описание реализации, скрин-шоты,* | | | | | | | | | | | | | |
| *список литературы, приложение (код программы)* | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | |
| Примерный объем пояснительной записки | | | | | | *10-15* | | | | страниц машинописного | | | |
| текста | | | | | | | | | | | | | |
| 5. Перечень графического материала ( с указанием обязательных чертежей и | | | | | | | | | | | | | |
| плакатов) | ------- | | | | | | | | | | | | |
|  | | | | | | | | | | | | | |
| 6. Консультанты | | | ------- | | | | | | | | | | |
|  | | | | | | | | | | | | | |
| 7. Дата получения задания: « 4 » сентября 2024 г. | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | |
| Руководитель | | | | |  | | |  | | | *Леонтьева Т.В.* | | |
|  | | | | | *(подпись)* | | |  | | | *(инициалы, фамилия)* | | |
| Задание принял к исполнению | | | | |  | |  | | | | *Абраамян А.М.,* | | |
|  | | | | | *(подпись)* | |  | | | | *(инициалы, фамилия)* | | |
|  | | | | | | | | | | | | 04.09.2024 | |
|  | | | | | | | | | | | | *(дата)* | |

**Содержание**

[**Введение** 4](#_Toc183955689)

[**Задание** 4](#_Toc183955690)

[**Описание реализации** 5](#_Toc183955691)

[**Ключевые компоненты программы** 5](#_Toc183955692)

[**Основные характеристики системы частиц** 6](#_Toc183955693)

[**Результат работы программы** 7](#_Toc183955694)

[**Заключение** 8](#_Toc183955695)

[**Список источников** 9](#_Toc183955696)

[**Код программы** 10](#_Toc183955697)

# **Введение**

Система частиц (particle system) — это технология моделирования сложных объектов, не имеющих четких геометрических границ. Такие анимационные объекты используются для моделирования дождя, огня, дыма, пузырьков газа в жидкости, осколков взрывающихся снарядов и тому подобных образцов объективной реальности.

Одна частица представляет собой точку с координатами в пространстве и привязанной к ней массой. Сцена строится из большого количества таких частиц, испускаемых эмиттерами (emitters), при этом для каждой частицы задается также начальная скорость, срок жизни и другие свойства.

Время в системе частиц дискретно. За один такт времени каждая из частиц может переместиться под влиянием внешних сил, определенных не для каждой частицы отдельно, а для всей системы в целом - например, силы гравитации. Поведение системы задается правилами поведения частиц в этой системе.

Количество частиц в системе может быть постоянным либо меняться со временем. Если количество частиц может меняться, то могут появляться (зарождаться) новые частицы или исчезать (умирать) старые.

Свойствами частицы могут быть, например, цвет, время жизни, размер, скорость и направление движения и т.д. Набор свойств определяет тип частицы. Параметры свойств могут меняться в течение жизни частицы, но сам набор свойств частицы остается неизменным. При зарождении частицы ее свойства инициализируются начальными значениями.

# **Задание**

Осуществить моделирование физического взаимодействия частиц с объектами в сцене:

1. Эмиттер – сфера
2. Обязательные параметры: прозрачность изменяется в зависимости от времени жизни. Остальные параметры устанавливаются и изменяются по вашему выбору.
3. След: присутствует, длина от 4 до 6
4. Столкновения: куб

# **Описание реализации**

Для реализации трехмерной сцены с системой частиц на языке Python использованы следующие библиотеки:

1. **Pygame**

Используется для управления окнами, обработки событий и создания интерфейса приложения. Модуль предоставляет возможности для взаимодействия с пользователем и обработки времени, что важно для анимации системы частиц.

1. **OpenGL.GL и OpenGL.GLU**

Эти модули позволяют работать с функциями OpenGL.

* OpenGL.GL: предоставляет базовые функции и константы для рендеринга 3D-графики.
* OpenGL.GLU: содержит функции высокого уровня, такие как создание перспективных проекций и работы с трёхмерными объектами.

1. **Numpy**

Используется для работы с векторами и вычислениями в трехмерном пространстве, такими как нормализация векторов, расчет расстояний и направление движения частиц.

1. **math и random**

Модули Python для выполнения математических операций и генерации случайных чисел, что необходимо для задания направления и скорости частиц.

## **Ключевые компоненты программы**

1. **Частицы (Particle)**  
Реализуют поведение отдельных частиц:

* Инициализация: задаются начальные позиция, скорость, направление движения и время жизни частицы.
* Обновление: Частицы обновляют свои координаты, учитывая:
  + гравитацию;
  + радиальное ускорение, зависящее от расстояния до эмиттера;
  + отражение от куба при столкновении.
* Отрисовка: Частица рисуется как точка в трехмерном пространстве с использованием OpenGL.

2. **Эмиттер частиц**

* Генерация частиц происходит из основания конуса. Направление движения частиц ориентировано в сторону куба, который служит ориентиром для их полета.
* Новые частицы создаются при удалении старых (превысивших время жизни).

3. **Объекты сцены**

* **Куб**: Представлен как объект столкновения. При попадании частицы в границы куба её скорость отражается по оси YYY. Куб визуализируется с гранями синего цвета и белой обводкой.
* **Конус (эмиттер)**: выступает источником частиц. Отрисовывается с помощью функции gluCylinder, а его основание обводится белой линией для визуального акцента.

4. **Главная программа**

* **Инициализация окна**: создается окно с использованием Pygame, а для визуализации настраивается OpenGL (камера, перспективы, тест глубины).
* **Цикл обновления**:
  + Рисуются куб и сфера.
  + Частицы обновляются и отрисовываются.
  + Проверяется пользовательский ввод (нап ример, завершение программы).
* **Обновление частиц**: Каждая частица перемещается в пространстве, а устаревшие частицы заменяются новыми.

## **Основные характеристики системы частиц**

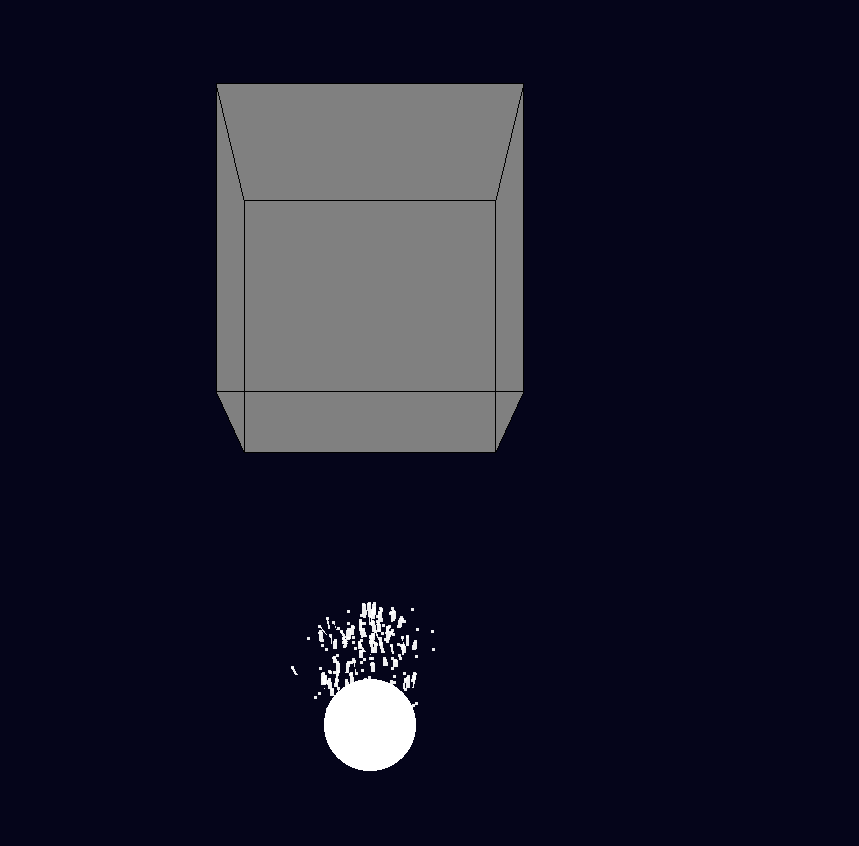
1. **Гравитация и радиальное ускорение**  
Частицы подвержены действию гравитации и ускоряются от эмиттера, что создает реалистичное движение.

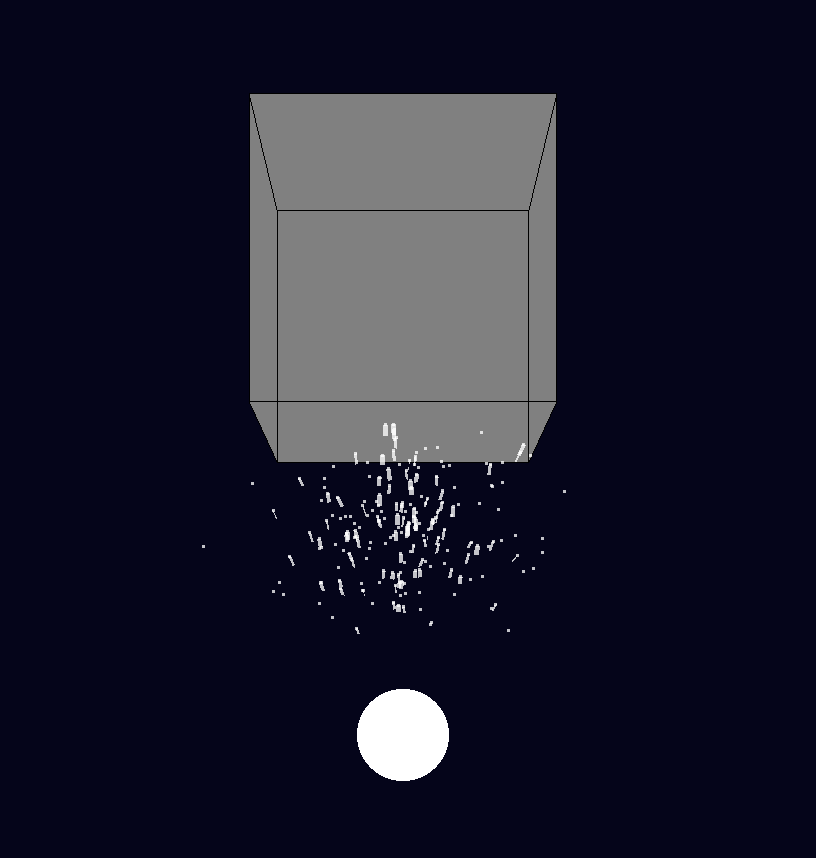
2. **Столкновения с кубом**  
При попадании в границы куба частицы отскакивают, что добавляет динамику в систему.

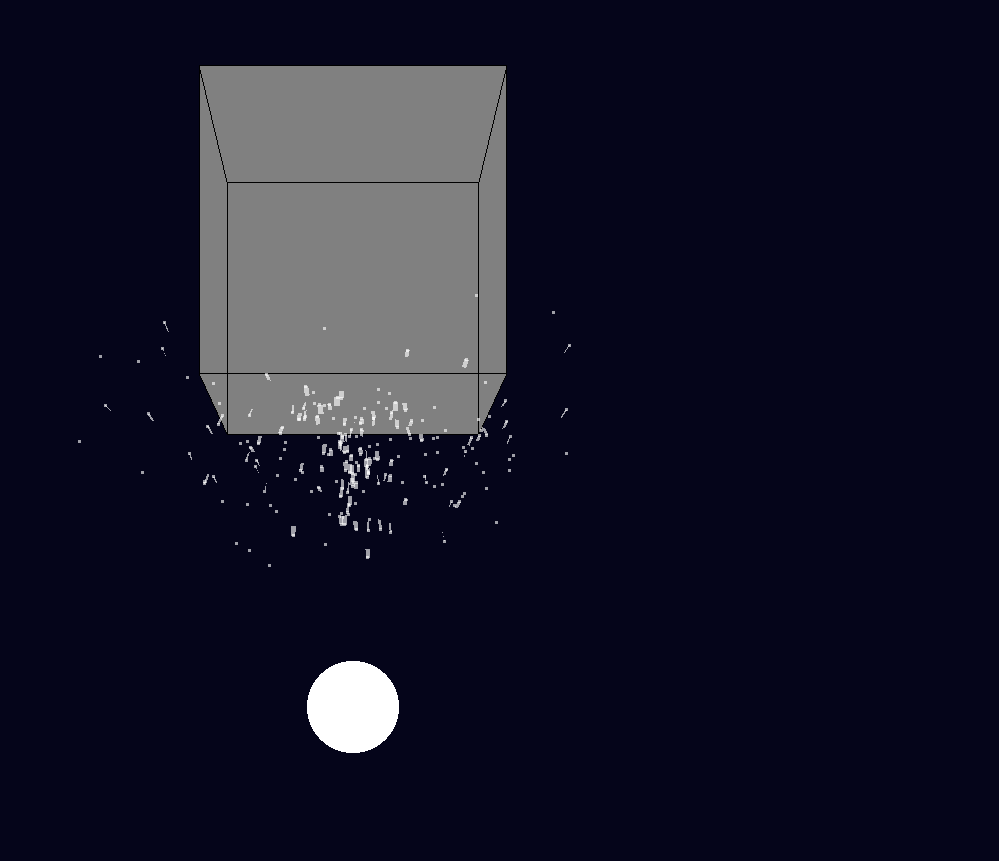
3. **Параметры частиц**  
Управляются их начальная скорость, размер, и время жизни, задаваемое при создании.

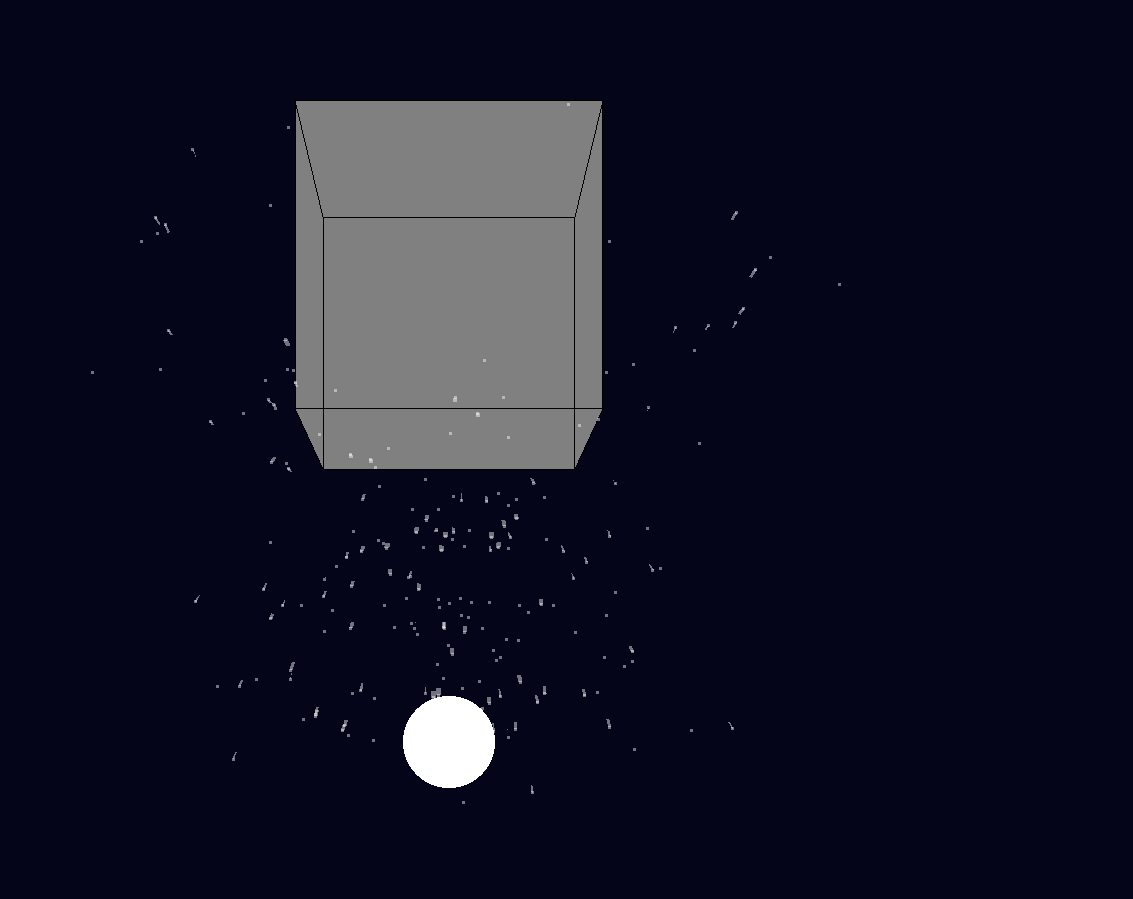
4. **Визуализация**  
Использование OpenGL позволяет создавать трехмерную графику с плавной анимацией.

# **Результат работы программы**









# **Заключение**

В рамках выполнения работы по моделированию физического взаимодействия частиц с объектами в сцене была создана программа на языке Python с использованием библиотеки OpenGL. В процессе разработки реализованы ключевые элементы системы частиц, включая эмиттеры, базовые параметры частиц, следы, аттракторы, а также учтена динамическая прозрачность, зависящая от скорости частиц. Работа позволила приобрести навыки использования графической библиотеки и моделирования поведения системы частиц.

# **Список источников**

1. PyGLM. Сайт Pypi.org // URL: <https://pypi.org/project/PyGLM/>
2. Официальная документация OpenGL // URL: <https://www.opengl.org/documentation/>
3. Официальная документация Pytho // URL: <https://docs.python.org/>

# **Код программы**

|  |
| --- |
| import math  import random  import numpy as np  import pygame  from OpenGL.GL import \*  from OpenGL.GLU import \*  from pygame.locals import \*  # Constants  SCREEN\_WIDTH, SCREEN\_HEIGHT = 1800, 1600  BACKGROUND\_COLOR = (0.02, 0.02, 0.1, 1)  # Emitting particles from the sphere's center  EMITTER\_POSITION = (0, 0, 0) # Sphere at the origin  EMITTER\_RADIUS = 1 # Radius of the sphere for particle emission  # Particle constants  MIN\_PARTICLES = 200 # Increased number of particles  MAX\_PARTICLES = 500 # Increased number of particles  LIFETIME = 5  PARTICLE\_SIZE = 1  TRACE\_LENGTH\_MIN = 1  TRACE\_LENGTH\_MAX = 2  GRAVITY = np.array([0, -0.01, 0])  # Cube for bump figure  CUBE\_SIZE = 3  CUBE\_POSITION = (0, 8 + CUBE\_SIZE / 2, 0)  class Particle:  def \_\_init\_\_(self, start\_position):  self.position = np.array(start\_position, dtype=float)  self.velocity = self.calculate\_initial\_velocity()  self.acceleration = GRAVITY  self.size = PARTICLE\_SIZE  self.age = 0  self.trace\_length = random.randint(TRACE\_LENGTH\_MIN, TRACE\_LENGTH\_MAX)  self.trail = []  def calculate\_initial\_velocity(self):  # Calculate the direction from the emitter (sphere) to the cube  direction\_to\_cube = (  np.array([CUBE\_POSITION[0], CUBE\_POSITION[1], CUBE\_POSITION[2]])  - self.position  )  direction\_to\_cube /= np.linalg.norm(direction\_to\_cube) # Normalize the vector  # Add some randomness to the direction  random\_angle = random.uniform(  -math.pi / 4, math.pi / 4  ) # Randomize angle slightly  rotation\_axis = np.array(  [random.uniform(-1, 1), random.uniform(-1, 1), random.uniform(-1, 1)]  )  rotation\_axis /= np.linalg.norm(rotation\_axis) # Normalize to get a unit vector  # Apply random rotation to the direction vector  rotation\_matrix = self.rotation\_matrix(  direction\_to\_cube, random\_angle, rotation\_axis  )  direction\_to\_cube = np.dot(rotation\_matrix, direction\_to\_cube)  # Return the velocity scaled by a random speed factor  return direction\_to\_cube \* random.uniform(3.0, 5.0)  def rotation\_matrix(self, vector, angle, axis):  # Compute a rotation matrix using the axis-angle formula  cos\_theta = math.cos(angle)  sin\_theta = math.sin(angle)  ux, uy, uz = axis  # Rotation matrix components  rotation\_matrix = np.array(  [  [  cos\_theta + ux \* ux \* (1 - cos\_theta),  ux \* uy \* (1 - cos\_theta) - uz \* sin\_theta,  ux \* uz \* (1 - cos\_theta) + uy \* sin\_theta,  ],  [  uy \* ux \* (1 - cos\_theta) + uz \* sin\_theta,  cos\_theta + uy \* uy \* (1 - cos\_theta),  uy \* uz \* (1 - cos\_theta) - ux \* sin\_theta,  ],  [  uz \* ux \* (1 - cos\_theta) - uy \* sin\_theta,  uz \* uy \* (1 - cos\_theta) + ux \* sin\_theta,  cos\_theta + uz \* uz \* (1 - cos\_theta),  ],  ]  )  return rotation\_matrix  def update(self, delta\_time):  self.apply\_gravity(delta\_time)  self.update\_position(delta\_time)  self.age += delta\_time  self.update\_trail()  def apply\_gravity(self, delta\_time):  self.velocity += self.acceleration \* delta\_time  def update\_position(self, delta\_time):  self.position += self.velocity \* delta\_time  def update\_trail(self):  if len(self.trail) >= self.trace\_length:  self.trail.pop(0)  self.trail.append(self.position.copy())  def draw(self):  self.draw\_particle\_point()  self.draw\_particle\_trail()  def draw\_particle\_point(self):  # Calculate transparency based on age  alpha = max(0, 1 - self.age / LIFETIME) # Transparency decreases with time  glColor4f(1.0, 1.0, 1.0, alpha) # White color with varying transparency  # Draw particle as a small round point  glPointSize(self.size \* 3) # Make particles smaller and more subtle  glBegin(GL\_POINTS)  glVertex3fv(self.position)  glEnd()  def draw\_particle\_trail(self):  # Draw the trail with GL\_QUAD\_STRIP for thickness but add color fading  glBegin(GL\_QUAD\_STRIP)  for i in range(len(self.trail) - 1):  pos1 = self.trail[i]  pos2 = self.trail[i + 1]  # Perpendicular vector for trail thickness (keep it slightly thinner)  perpendicular = np.cross(self.velocity, [0, 1, 0])  perpendicular = (  perpendicular / np.linalg.norm(perpendicular) \* self.size \* 0.05  )  left1 = pos1 - perpendicular  right1 = pos1 + perpendicular  left2 = pos2 - perpendicular  right2 = pos2 + perpendicular  # Color of the trail will fade from white to transparent as the particle ages  trail\_alpha = max(0, 1 - self.age / LIFETIME) # Fade the trail as the particle ages  glColor4f(1.0, 1.0, 1.0, trail\_alpha) # White trail with fading alpha  glVertex3fv(left1)  glVertex3fv(right1)  glVertex3fv(left2)  glVertex3fv(right2)  glEnd()  def create\_particle():  # Random position on the surface of the sphere  theta = random.uniform(0, 2 \* math.pi)  phi = random.uniform(0, math.pi)  x = EMITTER\_POSITION[0] + EMITTER\_RADIUS \* math.sin(phi) \* math.cos(theta)  y = EMITTER\_POSITION[1] + EMITTER\_RADIUS \* math.cos(phi)  z = EMITTER\_POSITION[2] + EMITTER\_RADIUS \* math.sin(phi) \* math.sin(theta)  return Particle((x, y, z))  def initialize\_particles():  particles = [create\_particle() for \_ in range(MIN\_PARTICLES)]  return particles  def update\_particles(particles, delta\_time):  for particle in particles[:]:  handle\_particle\_collision(particle)  particle.update(delta\_time)  if particle.age > LIFETIME:  particles.remove(particle)  particles.append(create\_particle())  def handle\_particle\_collision(particle):  # Cube collision detection and handling (improved bounce)  min\_x, max\_x = CUBE\_POSITION[0] - CUBE\_SIZE, CUBE\_POSITION[0] + CUBE\_SIZE  min\_y, max\_y = CUBE\_POSITION[1] - CUBE\_SIZE, CUBE\_POSITION[1] + CUBE\_SIZE  min\_z, max\_z = CUBE\_POSITION[2] - CUBE\_SIZE, CUBE\_POSITION[2] + CUBE\_SIZE  margin = 0.05 # Fine-tune this margin to see better interaction  # Check if the particle is near the cube's faces (considering the margin)  if (  min\_x - margin <= particle.position[0] <= max\_x + margin  and min\_y - margin <= particle.position[1] <= max\_y + margin  and min\_z - margin <= particle.position[2] <= max\_z + margin  ):  print(f"Particle collided with cube at {particle.position}")    # Handle collision along each axis separately  # Reflect particle's velocity and clamp position to cube's surface    # X-axis collision detection  if min\_x <= particle.position[0] <= max\_x:  particle.velocity[1] = -particle.velocity[1]    # Y-axis collision detection  if min\_y <= particle.position[1] <= max\_y:  particle.velocity[1] = -particle.velocity[1]    # Z-axis collision detection  if min\_z <= particle.position[2] <= max\_z:  particle.velocity[1] = -particle.velocity[1]  def draw\_cube():  # Set the cube's color  glColor3f(0.5, 0.5, 0.5) # Light gray color  glPushMatrix()    # Translate to the correct position of the cube  glTranslatef(CUBE\_POSITION[0], CUBE\_POSITION[1], CUBE\_POSITION[2])  # Draw the faces of the cube  glBegin(GL\_QUADS)  for face in [  # Front face  [(1, 1, -1), (-1, 1, -1), (-1, -1, -1), (1, -1, -1)],  # Back face  [(1, 1, 1), (1, -1, 1), (-1, -1, 1), (-1, 1, 1)],  # Left face  [(-1, 1, 1), (-1, -1, 1), (-1, -1, -1), (-1, 1, -1)],  # Right face  [(1, 1, -1), (1, -1, -1), (1, -1, 1), (1, 1, 1)],  # Top face  [(1, 1, 1), (-1, 1, 1), (-1, 1, -1), (1, 1, -1)],  # Bottom face  [(1, -1, -1), (-1, -1, -1), (-1, -1, 1), (1, -1, 1)],  ]:  for vertex in face:  glVertex3f(\*[CUBE\_SIZE \* x for x in vertex])  glEnd()  # Draw cube edges (outline)  glColor3f(0, 0, 0) # Black color for cube edges (frames)  glBegin(GL\_LINES)  for face in [  # Front face  [(1, 1, -1), (-1, 1, -1), (-1, -1, -1), (1, -1, -1)],  # Back face  [(1, 1, 1), (1, -1, 1), (-1, -1, 1), (-1, 1, 1)],  # Left face  [(-1, 1, 1), (-1, -1, 1), (-1, -1, -1), (-1, 1, -1)],  # Right face  [(1, 1, -1), (1, -1, -1), (1, -1, 1), (1, 1, 1)],  # Top face  [(1, 1, 1), (-1, 1, 1), (-1, 1, -1), (1, 1, -1)],  # Bottom face  [(1, -1, -1), (-1, -1, -1), (-1, -1, 1), (1, -1, 1)],  ]:  # Draw edges for each face by connecting vertices  for i in range(len(face)):  start = np.array(face[i]) \* CUBE\_SIZE  end = np.array(face[(i + 1) % len(face)]) \* CUBE\_SIZE  glVertex3fv(start)  glVertex3fv(end)  glEnd()  glPopMatrix()  def draw\_sphere():  # Draw the sphere at the emitter position  glPushMatrix()  glTranslatef(EMITTER\_POSITION[0], EMITTER\_POSITION[1], EMITTER\_POSITION[2])  # Set color of the sphere (e.g., blue)  glColor3f(1, 1, 1)  # Use gluSphere to draw the sphere  sphere = gluNewQuadric()  gluQuadricDrawStyle(sphere, GLU\_FILL)  gluSphere(sphere, EMITTER\_RADIUS, 32, 32) # 32 slices and 32 stacks for smoothness  gluDeleteQuadric(sphere)  glPopMatrix()  def initialize\_pygame():  pygame.init()  pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT), DOUBLEBUF | OPENGL)  pygame.display.set\_caption("course work")  gluPerspective(60, (SCREEN\_WIDTH / SCREEN\_HEIGHT), 0.1, 100.0)  glTranslatef(0.0, 0, -30)  # glRotatef(40, 0, 1, 0)  glEnable(GL\_BLEND)  glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA)  return pygame.time.Clock()  def main\_loop():  clock = initialize\_pygame()  particles = initialize\_particles()  running = True  while running:  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)  glClearColor(\*BACKGROUND\_COLOR)  delta\_time = clock.get\_time() / 1000  for event in pygame.event.get():  if event.type == QUIT:  running = False  # Draw the sphere at the emitter position  draw\_sphere()    # Draw the cube above the sphere  draw\_cube()  # Update and draw particles  update\_particles(particles, delta\_time)  for particle in particles:  particle.draw()  pygame.display.flip()  clock.tick(60)  pygame.quit()  if \_\_name\_\_ == "\_\_main\_\_":  main\_loop() |