Gravity Simulation Web Application

By:

Maria Kuprikova

Submitted to the Department of Computer Science and Mathematics in partial fulfillment of the requirements for the degree of Bachelor of Arts.

Purchase College

State University of New York

May 2023

Mentor: Professor Irina Shablinsky

Second Reader: Professor Milton Primer

Abstract

This research project aims to study key concepts of computational physics by creating a simulation of gravity, with a focus on its implementation and design of the simulation itself. The project involves building up a solid knowledge of physics and differential equations that are required for accurate simulations and exploring the limits of simulations compared to real-life experimentation. It takes the concepts of physics and programming and demonstrates their intersection, showing how physics simulations are important for scientists to better visualize and understand the world around us. By making a fun yet educational web application that can be used by a variety of people, I want to emphasize the importance of a well-designed user interface in creating a positive user experience. My ultimate goal is to demonstrate my expertise and capabilities in creating a complex program through this project.

Contents

List of Figures

Chapter 1: Introduction

Physics is the study of the natural world, focusing on the fundamental principles that govern the behavior of matter and energy. Programming, on the other hand, is the art of writing computer software to perform specific tasks or solve problems. These two fields intersect in many ways, with programming being an essential tool in modern physics research.

One area where physics and programming come together is in the simulation of gravity. Gravity simulations allow us to model the motion of celestial bodies, such as planets and stars, and study the complex interactions between them. With the help of programming, physicists can create accurate and detailed simulations that can provide insights into the workings of the universe and helping us understand phenomena that might be impossible to observe directly.

One fundamental aspect of creating such simulations involves a fundamental understanding of designing a User Interface (UI). User interface refers to the means by which users interact with a software application or a machine. It encompasses all the visual, auditory, and tactile elements that make up the user experience. The importance of a well-designed user interface cannot be overstated, as it has a direct impact on how users perceive and use a program.

A good user interface can make a program more intuitive and easier to use, reducing the learning curve for new users and making the overall experience more enjoyable. It can also increase productivity and efficiency, as users can quickly access the features they need and complete tasks with minimal effort.

Overall, a good user interface is essential for creating a positive user experience, improving productivity, and driving user engagement. As such, it is a critical component of any physics simulation and should be given careful consideration and attention during the design and development process.

The main goal of my research project is to study key concepts of computational physics by attempting to create a simulation of gravity. There are technically two parts to the research, implementation, and design. Firstly, I wish to understand the importance of modern physics and how simulations are made to provide simple representations and solutions to complex problems. Secondly, I want to create an interactive web program that simulates gravity and allows the user to play around with inputs to explore the different ways that gravity can function.

Through this project, I want to learn about the difficulty of creating accurate simulations that can be accurately used for study and practice. There are also many questions that can be asked after completing the program. Such as: What improvements can be made to the program? How accurate is the program? What are some of the limits of simulations over real-life experimentation? I will be working on building my knowledge of physics and differential equations required to create an environment that is able to produce a somewhat accurate description of natural phenomena. Over the course of my project, there will be an evolution to my program as I work through the difficult equations and formulas that will require time and effort to do correctly. With these expressions, I hope to demonstrate my heightened knowledge of the subject of computational physics as well as my capabilities in creating a complicated program/simulation.

Chapter 2: Literature Review

Physics and Gravity

Physics is a natural science that seeks to understand the physical world through observation, experimentation, and mathematical modeling. It focuses on describing the behavior of matter and energy, and how they interact with each other in the natural world. Physics seeks to identify and understand the laws that govern the behavior of the physical universe, such as the laws of motion, the laws of thermodynamics, and the laws of electromagnetism.

Computational physics is a field of study in physics that combines mathematics, computer science, and computational techniques to solve complex physical problems. It involves the development and application of numerical methods, algorithms, and computer simulations to model and analyze physical phenomena, ranging from the behavior of subatomic particles to the evolution of galaxies.

By simulating physical systems, we can predict how objects and concepts will behave under different conditions as well as save time and money by allowing researchers to test hypotheses without the need for expensive or dangerous experiments. Simulations can also provide visual representations of complex physical systems that are difficult to observe directly and can be used as educational tools to help students better understand complex physical concepts.

The importance of computational physics lies in its ability to provide insight into physical systems that are too complex to study experimentally or analytically. Computational models can simulate the behavior of systems under various conditions, predict outcomes, and provide a basis for designing experiments and making predictions.

Any good physics textbook will do the trick in teaching various concepts, but I've found the physics textbooks offered at openstax.com to be the most useful. The first step in this project is to study and understand the concept of relativity as stated previously. This starts with a thorough understanding of what force is. Force is a physical quantity that describes the interaction between two objects, which can cause a change in motion or deformation of the objects. In simpler terms, a force is a push or pull exerted on an object that causes it to move, change direction, or deform. The unit of force is the Newton (N) in the

International System of Units (SI). Gravity is a *natural* force that causes two objects to be attracted to each other. It is one of the four fundamental forces of nature, along with electromagnetism, the strong nuclear force, and the weak nuclear force. According to the theory of general relativity proposed by Albert Einstein, gravity is the result of the curvature of spacetime caused by the presence of objects with mass. When such an object is present, it warps the fabric of space-time around it, creating a curvature that causes other objects to appear to move toward the initial object as if it is attracted to it.

We can't see how gravity works exactly, but there are ways to help us visualize it. Imagine taking a flexible cloth and stretching it across on all sides until it becomes flat. Now, drop an object into the center of the cloth, since it has mass, it will sink down into the cloth, creating a dip in the cloth. If I were to throw another object with mass onto the cloth, that object would start to move towards the initial object due to the warping of the cloth as shown in Figure 1.1. This is the current model we use to visualize gravity based on the assumption that we live in a flexible space-time.

Figure 1. 1 A visual representation of gravity

Gravity is responsible for many important phenomena in the universe, including the orbits of planets around stars, the formation of galaxies, and the behavior of black holes. Gravity is the great creator, the constructor of worlds. Without gravity, the universe would be a very different place, and many of the structures we observe would simply not exist.

The idea of gravity has been around since ancient times, but our modern understanding of gravity is largely due to the work of Sir Isaac Newton, who developed the law of universal gravitation in the 17th century. Isaac Newton was an English physicist and mathematician who is widely regarded as one of the most influential scientists of all time. His discovery of the law of gravity was a breakthrough in science and paved the way for many other important discoveries and advancements in physics and astronomy.

Newton's work on gravity began in the late 1660s, when he was still a student at Cambridge University. He was inspired by the work of Galileo Galilei, an Italian philosopher who had discovered the laws of motion and the principles of inertia a few decades earlier. Newton discovered the law of gravity by building upon Galileo’s discoveries, and through a combination of theoretical calculations and experimental observations he realized that the same force that causes objects to fall to the ground also governs the motion of the planets in the solar system. Newton's breakthrough came in 1687, when he published his book, "Mathematical Principles of Natural Philosophy," also known as the "Principia." In this book, Newton outlined his three laws of motion and the law of universal gravitation.

The first law of motion describes inertia, which states that an object at rest will remain at rest, and an object in motion will remain in motion with a constant velocity, unless acted upon by an external force. The second law of motion is the mathematical equation that describes force. It states that the force applied to an object is equal to the object's mass times its acceleration, which is represented by the mathematical formula:

$$F = ma$$

The third law of motion states that for every action, there is an equal and opposite reaction. Along with these three laws, Newton also described the law of universal gravitation, which states that every particle in the universe attracts every other particle with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between them. This is mathematically written as:

$$F = G\frac{m_1 m_2}{r^2}$$

As stated before, understanding gravity is crucial for understanding the structure and evolution of the universe. Gravity is a fundamental force of nature that affects how all objects move. Understanding gravity allows us to predict and explain the motion of objects, from the smallest particles to the largest structures in the universe. Gravity is a crucial factor in space exploration, allowing us to launch spacecraft and satellites, navigate them through space, and land them on other planets.

The Nature of Code

"The Nature of Code" is a book written by Daniel Shiffman that explores the principles and techniques of coding simulations and generative art. The book uses examples written in the programming languages Processing and P5.js to explain concepts such as vectors, physics, and artificial intelligence. Throughout the book, Shiffman emphasizes the idea that many natural phenomena can be modeled and understood through the lens of code. He covers topics such as steering behaviors, particle systems, and genetic algorithms, and shows how they can be used to create simulations of natural systems like flocks of birds or mutual attraction.

Each chapter in the book includes code examples and exercises that allow readers to experiment with the concepts presented in the chapter. The book is overall divided into three sections: Foundations, Nature, and The Mind. The Foundations section introduces basic concepts of programming and mathematics, while the Nature section explores how these concepts can be applied to model natural systems. The Mind section covers more advanced topics, such as neural networks and genetic algorithms.

The focus here in my project will be on the second section, Nature. Part two of "The Nature of Code" explores the specifics of using code to simulate natural systems. This part of the book builds on the foundation laid in part one, which introduced basic concepts such as vectors, forces, and particle systems. The chapters in part two cover a range of topics, including physics simulations, fractal geometry, and cellular automata.

Part 2 starts at Chapter 3, which explores the principles of oscillation and harmonic motion, and shows how to use code to create simulations of pendulums, springs, and other oscillating systems. Chapter 4 focus on forces, explaining how to use vectors to represent forces and motion in a simulation. It covers topics such as gravity, friction, and wind resistance, and shows how to create simulations of bouncing balls and flocking behavior in birds. Chapter 5 delves into the usage of Physics Libraries. It introduces the concept of using physics libraries to simplify the process of creating simulations. It shows how to use libraries such as Box2D and Toxiclibs to create simulations of physical systems such as fluids and soft bodies. Chapter 6 covers Autonomous Agents, which are individual entities that can make decisions and act independently within a simulation, such as steering behaviors, pathfinding, and swarm intelligence. Chapter 7 is about Cellular Automata, grids of cells that can change state based on simple rules. It shows how to use code to create simulations of complex patterns, such as the Game of Life. Lastly, chapter 8 concludes by discussing Fractals, the study of patterns that repeat at different scales. It explains how to use code to create simulations of fractals such as the Mandelbrot set and the Sierpinski triangle.

Let’s take a closer look at chapter four of "The Nature of Code”. This chapter covers the concept of forces and motion in simulations and shows how to use vectors to represent them. The chapter begins with an introduction to the principles of physics, including Newton's laws of motion and the concept of force.

The chapter covers a range of topics related to forces and motion, including gravity, wind resistance, and friction. It shows how to use code to create simulations of bouncing balls, particle systems, and flocking behavior. One of the key concepts introduced in the chapter is the idea of a "force accumulator", which is a variable that accumulates forces applied to an object over time. The force accumulator is used to calculate the object's acceleration, velocity, and position based on the forces acting on it. The chapter also explores the concept of "drag", which is the force that opposes motion through a fluid. It shows how to use code to simulate drag and create more realistic simulations of physical systems.

Daniel Schiffman implements these concepts using Euler’s method of integration. Euler's method is a technique that is commonly used to simulate the motion of objects under the influence of gravity in computer programs. In the context of simulating gravity, the method involves updating the position and velocity of an object at each time step based on its current position, velocity, and acceleration. The acceleration is calculated based on the gravitational force acting on the object due to other objects in the system.

To use the Euler's method to simulate gravity, we first need to define the position, velocity, and acceleration of each object in the system. We can then calculate the gravitational force acting on each object due to the other objects in the system using Newton's Law of Universal Gravitation. Once we have calculated the gravitational force acting on each object, we can update the acceleration, velocity, and position of each object using the following equations:

Acceleration: $a = \frac{F}{m}$

Velocity: $v = v + a(dt)$ *Where dt is the time step size.*

Position: $p = p + v(dt)$

These equations update the position and velocity of each object based on its current position, velocity, and acceleration, as well as the time step size. **(Figure 1.2)** By iterating through these calculations at each time step, we can simulate the motion of the objects under the influence of gravity. Schiffman uses the implementation of Euler's method to update the position of his object relative to the forces acting upon it.

```
update() {

  // let mouse = createVector(mouseX, mouseY);
  // this.acc = p5.Vector.sub(mouse, this.pos);
  // this.acc.setMag(0.1);

  this.vel.add(this.acc);
  this.pos.add(this.vel);
  this.acc.set(0, 0);
}
```

Figure 1. 2 Snippet of the accumulator code

It's worth noting that Euler’s method is a relatively simple and straightforward numerical integration technique, but it can have some drawbacks in terms of accuracy and stability, particularly for complex systems or systems with *large* time steps. Other numerical integration techniques, such as the Runge-Kutta method, may be more accurate and stable for certain types of simulations.

The Runge-Kutta method is a more accurate and stable numerical method compared to Euler's method, because it considers intermediate values of acceleration over the time step, resulting in more accurate predictions of the future positions of objects. However, for the purposes of simplicity and to not focus on the implementation Schiffman uses Euler's method for his simulations.

Creating User Interfaces

A user interface (UI) is the graphical or visual part of a software or digital product that enables a user to interact with it. It includes all the elements, such as icons, buttons, menus, and forms that users interact with on a screen, as well as the way in which these elements are arranged and presented. The primary purpose of a user interface is to provide users with an intuitive and user-friendly way to interact with software or a digital product, enabling them to complete tasks and achieve their goals efficiently and effectively. An effective UI should be easy to navigate, provide clear feedback, and be visually appealing, as well as reflecting the branding and values of the product. Designing a user interface can be a complex and challenging process that requires careful consideration of many different factors.

The first step to creating a user interface is understanding the user's needs. The programmer needs to understand the users' preferences to create a UI that is intuitive, easy to use, and effective. This involves conducting user research and testing to gather feedback on the design. The UI will also need to be visually appealing, and the design should be consistent. Achieving the right balance between aesthetics and functionality can be a challenge, as a design that is too visually complex may be difficult for users to navigate.

A UI may have many different features and functions, each with their own UI elements and interactions. Managing this complexity can be a challenge, as a cluttered UI can make it difficult for users to find what they need. Meeting accessibility requirements is also a vital aspect of user design, and one often forgotten. The UI must be accessible to all users, including those with disabilities. However, meeting accessibility requirements is a challenge, as it requires designers to consider a wide range of user needs and design for different types of disabilities.

When it comes to physics simulations, the user interface is particularly important, as it allows users to visualize and interact with the simulation, making it easier to understand and manipulate **(Figure 1.3).** For example, in a physics simulation program, a well-designed user interface could include intuitive controls for adjusting the simulation parameters, visual feedback on the state of the simulation, and clear indications of the simulation's outputs. This would allow users to easily adjust and experiment with different simulation scenarios, and to quickly see the results of their changes.

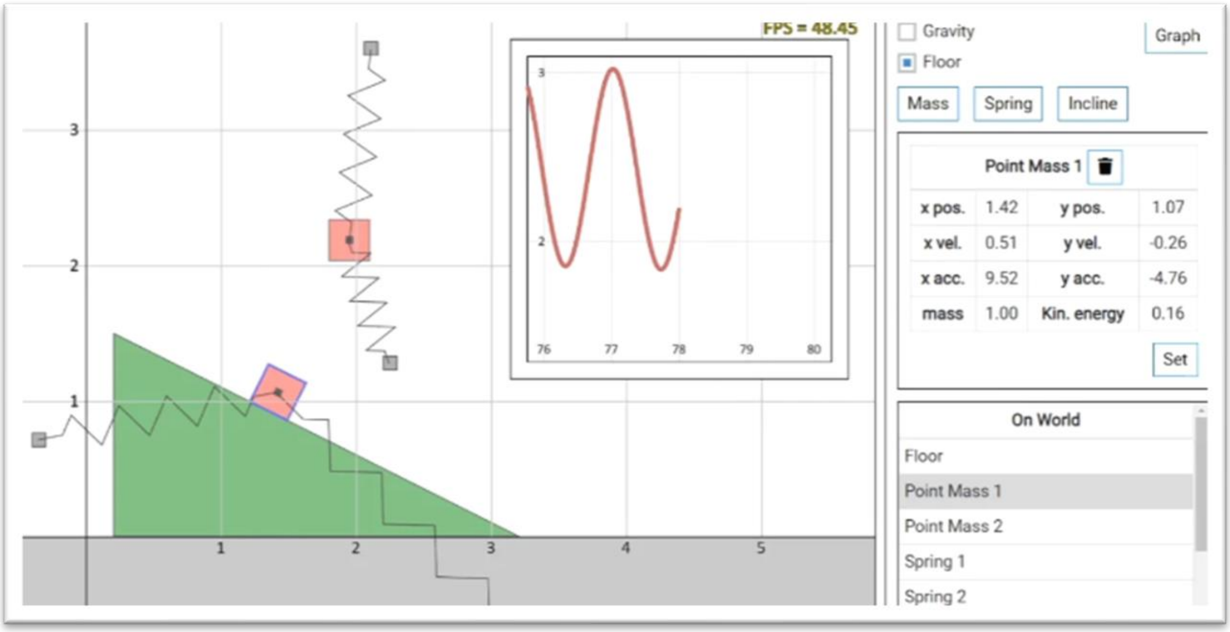

Figure 1.3 An example of a physics simulation program display.

Physics simulations are important in software development as they allow developers to model and understand the behavior of physical systems in a virtual environment. These simulations can be used to test and optimize designs, simulate real-world scenarios, and predict the behavior of complex systems.

For example, physics simulations are commonly used in video game development to create realistic game mechanics and interactions, such as collisions, gravity, and projectile motion. They can also be used in engineering and design fields to simulate the behavior of physical systems, such as fluid dynamics, structural mechanics, and thermal dynamics.

Overall, designing a UI requires careful consideration of many different factors, and it can be a complex and challenging process. However, by understanding user needs, balancing aesthetics, and functionality, ensuring consistency, dealing with limited screen space, managing complexity, and meeting accessibility requirements, designers can create UIs that are intuitive, effective, and easy to use.

Chapter 3: Research Methodology

Creating a Program and Implementing Gravity

Dogma of the Program

The two goals of this part of the project are to create a playground for modeling gravity as it appears on the surface of a planet, and another model that shows gravitational attraction as it appears in space. To understand how to create these models, we will take a deeper look at the laws of gravity laid out by Isaac Newton and examine how gravity is implemented in David Shiffman's book, "The Nature of Code". But first, we must understand the set of rules we need to follow if we want an accurate representation of gravity in a program.

As was described in the above literature review, The three laws of motion plus the law of gravitational attraction will be the dogma of my program. In other words, these principles provide an accurate representation of gravity in my simulation.

Framework of the Program

The tool I will be using to write my program in a web application called Angular, which is an open-source, front-end web application development framework maintained by Google. It is built on TypeScript, a superset of JavaScript, and is designed to help developers build dynamic, responsive web applications with ease. It uses a component-based architecture, where each component is a self-contained block of code that defines a part of the user interface and the associated logic. Components can communicate with each other through inputs and outputs, allowing for a modular and flexible design.

This framework contains many features that I will be using to my advantage throughout the project. First, Angular allows developers to establish a two-way data binding between the user interface and the data model, making it easy to keep the view and the data in sync. Its templating system allows developers to create dynamic, responsive user interfaces using HTML, CSS, and Angular-specific directives. It also has a dependency injection system making it easy to manage the dependencies of a component and to swap out components for testing or other purposes. Lastly, Angular provides tools for unit testing, integration testing, and end-to-end testing, making it easy to ensure that your application is functioning as expected.

Using Angular as a framework allows me to waste less time writing my own website functionality and stylesheets and instead focus on making the core aspect of my program, which will be the gravity simulation.

I will also be using GitHub to store my project. GitHub is a web-based platform that allows developers to collaborate on software development projects. It provides a platform for version control, code collaboration, and project management. With GitHub, developers can create and share repositories that contain code, documentation, and other resources related to their project.

Angular uses the Git version control system, which allows developers to track changes to their code over time. This system makes it easy to collaborate on projects with multiple developers and keep track of changes made to the codebase. GitHub also provides features like issue tracking, which allows developers to manage bugs and feature requests, and pull requests, which allow developers to propose changes to the codebase and have them reviewed by other developers before merging them into the main codebase.

Although I am working on my project on my own, for good practice I will still be using GitHub to save and monitor my program.

HTML Canvas

I will be using HTML Canvas to create the graphics of the gravity simulation in my program. HTML canvas is an HTML element that allows for dynamic, interactive graphics to be created and displayed on a web page. The canvas element itself is just a container for graphics, similar to an image tag, but unlike an image tag, the canvas can be dynamically modified with JavaScript to create animations, games, data visualizations, and other types of interactive content.

The canvas element provides an Application Programming Interface (API) that can be used to draw shapes, lines, text, and images on the canvas. An API works to provide a service to other pieces of software, which in this case is the canvas element service to my web application. This API includes methods for setting the fill and stroke colors, drawing paths and shapes, and applying transformations to the canvas.

In order to use the canvas element, you need to create a new canvas element in your HTML code and give it a unique ID. Then, you can use JavaScript, or in my case TypeScript, to access the canvas element, create a 2D or 3D rendering context, and begin drawing on the canvas. For my program, I will be manipulating the canvas using a class called IScene. It will hold all the functionality of my canvas element.

```
export interface IScene {
    ctx: CanvasRenderingContext2D;
    objects: OrderedMap<string, ISceneObject>;
    add(obj: ISceneObject): IScene;
    remove(name: string): IScene;
    hide(name: string): IScene;
    show(name: string): IScene;
    hideAllButOne(name: string): IScene;
    updateByKey(name: string, obj: ISceneObject): IScene;
    update(obj: ISceneObject | ISceneObject[]): IScene;
    updateWithCondition(condition: (obj: ISceneObject) => boolean, map: (obj: ISceneObject) => ISceneObject): void;
    clear(): IScene;
    postCalc: (() => void) | null;
    draw(): void;
    resize(): void;
    X(x: number): number;
    Y(y: number): number;
    worldX(x: number): number;
    worldY(y: number): number;
    scale: number;
    VisibleWorldHeight: number;
    width: number;
    height: number;
    gravity: number;
    elasticity: number;
    friction: number;
    mode: AppMode;
    inPause: boolean;
    showVelocityVector: boolean;
}
```

Figure 1.4 IScene class interface from my gravity-sim program

However, there is one limitation of the canvas element that needs to be addressed, the fixed size of the canvas. When a canvas is initialized, it's height and width values are set, meaning that when the window of the web page resizes, the canvas element will remain the same. In order to have a better viewing experience, I determined that having the canvas be responsive to the resizing of the screen would be a necessary feature to add. The canvas size is not constant and can be manipulated, however, the images within the canvas would also be resized and lose resolution as they get large. The images become increasingly pixelated as the canvas size increases.

The solution I found to this issue is creating a class that is used for rendering graphics on an HTML5 canvas, with a constructor that is initializing the canvas context and some scene properties. Within this class is a resize() method that is responsible for updating those properties when the canvas size changes. This way, the resolution of the image remains the same no matter the size of the canvas. Along with this approach, I created a separate app component that created the responsive design of the canvas element.

This method sets the size of the HTML canvas element to match the size of its parent element. The method gets a reference to the parent element of the canvas using the nativeElement property, which is a reference to the actual HTML element in the Document

Object Model (DOM) of the application. The Document Object Model is a language-independent interface that treats an HTML document as a tree structure wherein each node is an object representing a part of the document. Then, it uses the getBoundingClientRect() method to get the size and position of the parent element relative to the viewport. Finally, it sets the width and height properties of the nativeElement property of the canvas object to the width and height properties of the rect object, respectively. This setting ensures that the canvas element always fills the size of its parent element (the web page).

Gravity Implementation

One key feature of my program is that it uses the concept of polymorphism, which is a part of object-oriented programming (OOP). Polymorphism refers to the ability of objects to take on different forms or behave in different ways depending on the context in which they are used. In OOP, polymorphism is typically achieved through inheritance and method overriding. For example, in a class hierarchy where a subclass inherits from a superclass, the subclass can override methods defined in the superclass, allowing it to provide its own implementation of those methods. This means that different objects of the same class can behave differently depending on which subclass they belong to. For example, a Shape class might have subclasses like Circle, Square, and Triangle, each of which has its own implementation of a draw() method.

In my application, I use polymorphism to create a baseObject class that acts as base object model that can be placed into the scene of the canvas. A class such as the Ball class can inherit the properties of baseObject in order to be put into the scene, but if I wish to add in another object, I can always create a new class that inherits the baseObjects properties as a base.

Another important feature to note is the use of interfaces. In programming, an interface is a contract or agreement between two or more parts of a system. It defines a set of rules, requirements, and behaviors that one object or component must follow to interact with another object or component. An interface specifies a set of methods, properties, and events that an implementing class or object must provide, without defining how those methods should be implemented.

In other words, an interface is a blueprint for classes or objects that need to interact with each other. It defines a common set of methods and properties that can be used by different classes or objects, allowing them to communicate and work together in a consistent and predictable manner. By adhering to the rules and requirements of an interface, different parts of a system can be designed and implemented independently, without needing to know the details of each other's implementation. This makes interfaces a powerful tool for building modular, extensible, and reusable software components which will be helpful when creating this program.

Implementing gravity in code involves simulating the effects of gravitational force on objects in a virtual environment. So, the first step in my program is to create an object with dynamic values that can be manipulated via an input mechanism as well as a series of methods for interacting with the object. In other words, I wanted to create objects on the screen where I can change the attributes of it manually. This object will be affected by gravity and therefore should have a position and a mass. The methods will mostly be centered around drawing the object in the canvas element and moving the object's position on the screen.

```
export interface ISceneObject {
    name: string;
    enabled: boolean;
    position: IPoint;
    trace?: boolean;
    trace_limit?: number;
    delta(scene: IScene): void;
    collide(scene: IScene): void;
    draw(scene: IScene): void;
    draw_trace(scene: IScene): void;
    pan(p: IVector): void;
}
```

Figure 1.5 Object Class Interface

There are various classes derived from this class, such as Ball and BaseObject. The BaseObject class is responsible for the positioning of the object in the canvas and the Ball class implements the BaseObject's class as well to create, draw and update the "ball" that appears on the screen.

To move the object according to the gravitational force acting upon it (as well as other forces), we will have to use the concept of vectors to create a Vector Class. **(Figure 1.6)** This class will provide the necessary functionality to manipulate the objects movement and perform vector math.

```
export interface IVector {
    x: number;
    y: number;
    add(v: IVector) : IVector;
    sub(v: IVector): IVector;
    mul(n: number): IVector;
    div(n: number): IVector;
    magnitude: number;
    normalize(): IVector;
    dir(direction: number): IVector;
    dotProduct(v: IVector): number;
    crossProduct(v: IVector): number;
    toString(): string;
    isNull: boolean;
    angle(): number;
}
```

Figure 1.6 Vector class interface

With these two classes, we can start by first implementing gravity as it would appear on the surface of a planet. We will have to define the gravitational force that will be acting upon the object. In this case, the object will be pulled down to the ground since the gravitational attraction will point to the center of mass of the planet. Because I want to manipulate the simulation, the value of gravity will not be fixed so the user can manipulate its strength at will.

After calculating the gravitational force acting on each object, we can use Newton's second law of motion to calculate the acceleration of the objects due to that force, as shown in the figure below.

```
private update(scene?: IScene): void {
    if (scene && scene.gravity && scene.mode === AppMode.EarthGravity) {
        this.y_velocity -= scene.gravity;
    }
```

Figure 1.7 The objects velocity changed by acceleration (gravity pointing "down")

Finally, we can update the position and velocity of each object based on its current position, velocity, and acceleration. You can use numerical integration techniques like Euler's method to update the position and velocity of each object at each time step. This can be seen in the figure below.

```
this.position.x += this.x_velocity;
this.position.y += this.y_velocity;
```

Figure 1.8 Updating the position based on velocity

At this point we have an environment with only gravity. However, if we were to run the program as it is, the object that is affected by gravity will start moving down and never stop. This is because there are no boundaries on the canvas. So, the next step would be to add “walls” to the program.

```
private testWalls(scene: IScene): void {
    const bw = this.bounds.width / 2;
    const bh = this.bounds.height / 2;

    if (scene.X(this.position.x + bw) > scene.width) {
        this.x_velocity = -1 * this.x_velocity;
        this.position.x = scene.width / scene.scale - bw - 1;
    } else if (scene.X(this.position.x - bw) < scene.X(0)) {
        this.x_velocity = -1 * this.x_velocity;
        this.position.x = bw + 1;
    } else if (this.position.y > scene.VisibleWorldHeight) {
        this.y_velocity = -1 * this.y_velocity;
        this.position.y = scene.VisibleWorldHeight - bh - 1;
    } else if (scene.Y(this.position.y - bh) >= scene.Y(0)) {
        if (scene && scene.gravity && scene.elasticity && scene.mode === AppMode.EarthGravity) {
            this.y_velocity = -1 * this.y_velocity * scene.elasticity;
        } else {
            this.y_velocity = -1 * this.y_velocity;
        }
        if (scene && scene.gravity && scene.friction && scene.mode === AppMode.EarthGravity) {
            this.x_velocity = this.x_velocity - (this.x_velocity * scene.friction);
        }
        this.position.y = bh + ((scene.mode === AppMode.SpaceGravity) ? 1 : 0);
    }
```

Figure 1.9 TestWalls method in class Ball

In the figure above, I have created a private method used by the class Ball to test whether the object has hit a “wall” which is simply the edges of the visible canvas. If it has, the velocity will be reversed. There are also two conditions accounting for the object's elasticity and environment's friction. This method for detecting the boundaries of the canvas screen can also be used to detect collisions in the environment.

In programming, an elastic collision refers to a collision between two objects where there is no loss of kinetic energy. This means that the total kinetic energy before the collision is equal to the total kinetic energy after the collision. When two objects collide in an elastic collision, their velocities are exchanged, but their total energy remains constant.

Elastic collisions can be contrasted with inelastic collisions, where some of the kinetic energy is lost during the collision, usually as heat or sound. In programming, inelastic collisions are often used to simulate more realistic collisions, such as when a ball bounces on the ground or a car crashes into a wall.

To implement an elastic collision in programming, one needs to calculate the velocities of the objects before and after the collision using the laws of conservation of momentum and energy. The velocities of the objects can then be updated based on these calculations to simulate the collision. One addition to this collision is also determining the angle of the collision and the angle at which the objects will move away from each other.

The collision angle can be calculated using trigonometry. It is measured between the line of centers of the two objects at the point of collision and the tangent to the point of collision on the surface of each object.

These are the main aspects of the physics engine I have created in the canvas. One additional detail I added is a “trace”. I’ve implemented a method called draw trace which draws a trace of the path of a moving object on the canvas of the scene.

The method iterates over each point in an array of trace points. For each point, it checks if the point is within the bounds of the screen. If the point is out of bounds, the visible flag is set to false, and the iteration proceeds to the next point. If the point is visible, the method starts a new path by calling ctx.beginPath(), then draws a small circle centered on the point using ctx.arc() with a radius of 1.

The opacity of the circle is determined by the value of alpha, which is calculated as the ratio of the index of the current point to the minimum of this.trace limit and the length of trace points. If this.trace limit is zero or negative, alpha is set to 1 which means the circles are fully opaque. This creates a fading effect on the end of the trace, and the length of the trace is determined by a slider where the user can set the value.

With all of this set up, we can move on to programming gravity as it would appear in space. The main difference between gravity as it would appear on the ground and gravity in space is that gravity is no longer pointed down and is instead an attractive force between objects. For this case we will have to use Newton’s law of gravitational attraction.

We need to calculate the acceleration vector of a ball object in a scene based on the gravitational forces acting on it due to other balls being present on the canvas. The acceleration vector will then be used to update the velocity of the ball object.

We first initialize an acceleration vector variable to a new Vector object with x and y components both set to 0. It then uses the reduce() method (function provided in node_modules under immutable.d.ts) to iterate over all the objects in the scene's objects array. For each object that is not the same as the current ball object and is an instance of the

Ball class, it will calculate the gravitational force between the current ball and the other ball using the formula:

$$F = G\frac{m_1 m_2}{r^2}$$

G is the gravitational constant, m1 and m2 are the masses of the balls, and r is the distance between their centers.

The code first calculates the displacement vector between the two balls using the move() method of the Vector class. This method returns a new Vector object initialized with the horizontal and vertical components of the displacement vector. This Vector object represents the direction and magnitude of the displacement vector between the two points passed as arguments. It then calculates the strength of the gravitational force using the formula mentioned above, and scales the displacement vector by the strength of the force divided by the mass of the current ball to get the acceleration vector acting on the ball.

The isNull property is set to a variable named acc, which we use to check if this is the first acceleration vector being added to the acc variable. If it is, then the current acceleration vector is set as the new value of acc. Otherwise, the current acceleration vector is added to the existing acceleration vector stored in the acc variable. Lastly, the updated x_velocity and y_velocity of the current ball are calculated by adding the x and y components of the acc vector to the current x_velocity and y_velocity respectively.

To get a better understanding of how these vectors appear, I will be creating an option to show an arrow connected to the object, pointing in the direction the specific vector is pointing and the length will represent the magnitude or strength of that vector. For this we will want to draw the velocity and acceleration vectors on the canvas.

We need to calculate the magnitude of the velocity vector by multiplying the radius of the object by its speed and clamping the result between 10 and 100. We then create a new Vector object using the x_velocity and y_velocity properties of the object, normalize it, and multiply it by the magnitude calculated earlier to get a vector representing the velocity with the correct magnitude. Then we draw a line from the object's current position to a point offset by the velocity vector and add an arrowhead to the end of the line to indicate its direction. Then we repeat the steps above for the acceleration vector, only this time, the color of the arrow will change to better differentiate between the two vectors.

The simulation now has a functional gravitational system in place, the next step is to be able to freely manipulate values and the visual aspect of the simulation. For this, we will need a user interface.

Designing a User Interface

Now that we have a working gravity simulation in place, we will need to build an interface that will allow us to interact with the program. One of the key features of the Angular framework is that it provides a set of tools and components for building user interfaces quickly and easily.

Angular uses declarative HTML templates to define the structure and content of the UI. This makes it easier for developers to create UIs by separating the UI logic from the business logic. It also uses two-way data binding, which means that changes to the UI are automatically reflected in the underlying data model, and vice versa.

Angular also provides a set of built-in directives that can be used to add behavior and functionality to the UI as well as using components to encapsulate UI elements and behavior into reusable and modular units. Directives can be used to manipulate the DOM, handle user events, and perform other tasks. Components on the other hand are self-contained and can be easily reused across different parts of the application. This makes it easier to create complex UIs by composing smaller, reusable components. All of these features are very useful and greatly reduce the time spent programming.

First, we create a mat-toolbar component that makes a toolbar at the top of the page. It includes a title and three buttons for toggling between different views of the simulation: EarthGravity, SpaceGravity, and About. The title includes the name of the application, which is "Maria's Gravity Sim" and the current mode of the page, which is determined by a variable called mode. The text in the title changes dynamically based on the value of mode using a ternary operator that checks the value of mode against three possible values; AppMode.EarthGravity, AppMode.SpaceGravity, and AppMode.About.

The three buttons in the toolbar are created using the mat-icon-button component and are labeled with icons from the Font Awesome icon library. Each button is associated with a different mode of the application and calls a function toggleMode when clicked, passing in the corresponding mode as an argument.

Next, we create a mat-drawer-container component that creates a container for the main interactive content of the application, which is displayed in a drawer that slides out from the side of the page when a view is selected. It contains an autosize attribute that makes the drawer adjust its size automatically based on the content inside it. We use the directive *ngIf to ensure that the drawer is only displayed when the current mode is not equal to AppMode.About.

Inside the mat-drawer-container, there are two instances of the mat-drawer component, each with a different mode. One for earth gravity and another for space. The *ngSwitch directive is used to display the appropriate drawer based on the current value of mode. Each drawer contains a set of sliders and labels for adjusting various parameters of the simulation, such as the number of balls, the strength of gravity, and the elasticity of objects. These sliders are created using the mat-slider component and are bound to corresponding variables using the ngModel and ngModelChange directives.

On the bottom of the page, there is a footer, and it contains a group of buttons and a slide toggle. The *ngIf directive is used to conditionally render the entire <div> element based on whether the mode variable is not equal to AppMode.About.

Inside the footer, there are four button elements. The first button has a label "Start Simulator" and it, as the name suggests, starts the simulation in the canvas. The second and third buttons have labels "Pause Simulator" and "Resume Simulator" respectively. They control the state of the simulation in the canvas. When the simulation is running, the user has the ability to click “Pause Simulation” which will freeze the canvas and all of the elements within it. When this happens, the text in the button changes to “Resume Simulation”, and when the user presses it, the simulation will resume.

The fourth button has a label "Stop Simulator" and it completely stops whatever is happening in the canvas and clears the screen.

After the buttons, there is a <mat-slide-toggle> element that toggles the value of the isSideNavOpen variable when it is clicked. The [(ngModel)] directive is used to bind the state of the slide toggle to the isSideNavOpen variable. When it is switched on, the side navigation containing all the sliders will slide closed to the left.

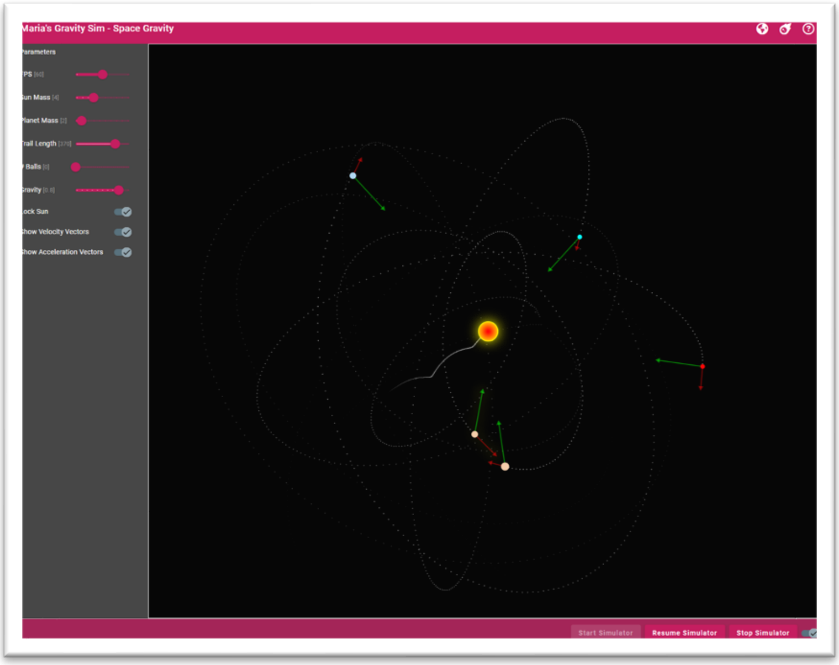

Figure 1. 10 Space Gravity View

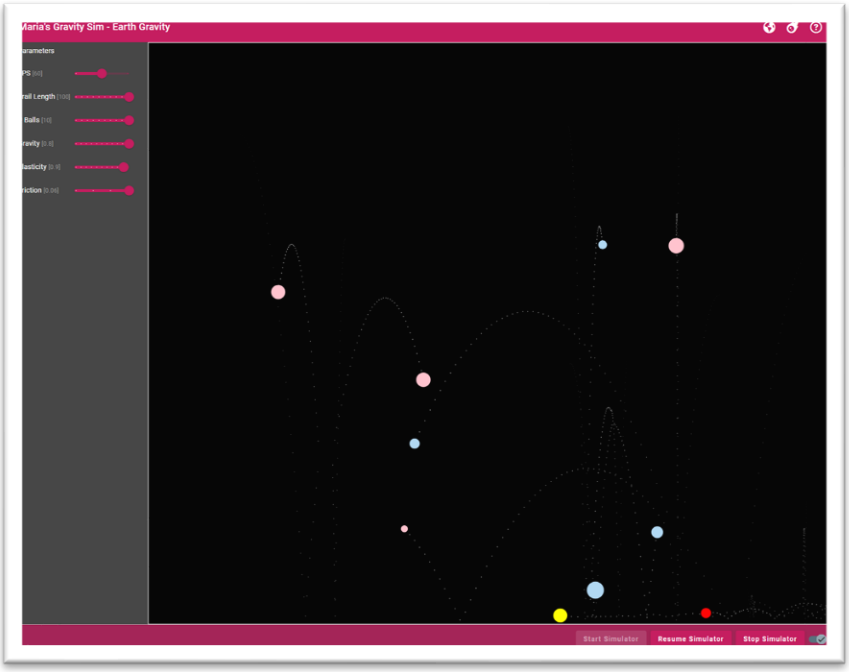

Figure 1.11 Earth Gravity View

Besides the visual elements of the graphic user interface, there is one other interface function built into the UI. For both earth and space gravity, the user has the ability to click anywhere in the canvas to add an object. The object will retain the pre-set parameters set by the user in the nav-bar, but will have some randomization for color and size of the object.

To do this, we need to add event listeners for mouse events to the canvas element. When the mouse is clicked down, the start position of the drawing is saved in a start_drawing variable as a Vector object with the x and y coordinates of the mouse's clientX and clientY properties. When the mouse button is released, a new Vector object end_drawing is created from the clientX and clientY properties of the mouse event.

If the symState variable is not equal to SymState. Playing or if start_drawing is null, then nothing happens and the function returns. But, if the symState is equal to SymState.Playing and start_drawing is not null, then some calculations are performed to determine the angle of the line drawn between start_drawing and end_drawing. The getBoundingClientRect() method is used to get the position of the canvas on the page, and the position of the start point is calculated relative to the canvas element. The addRandomBall() method is then called to add a new ball object to the scene, passing in the position and angle of the line drawn.

User interfaces are valuable because they enable users to interact with software applications in a way that is intuitive, efficient, and effective. By designing UIs that are user-friendly, accessible, and consistent, developers can create applications that are both powerful and easy to use, enhancing the overall value and usability of the software.

Chapter 4: Research Findings and Discussion

Researching how gravity works is fundamental to programming gravity because it requires an understanding of the underlying physics principles that govern gravity in the real world. By researching how gravity works, a programmer can gain an understanding of the mathematical equations and concepts that describe gravity and its effects. In programming, gravity is typically simulated using a set of equations that describe the behavior of objects under the influence of gravity. These equations consider factors such as mass, distance, and acceleration, which are all governed by the laws of physics. Therefore, to accurately simulate gravity in a program, a programmer needs to have a strong understanding of the physics principles that govern gravity.

While the underlying physics principles of gravity remain the same, the implementation of gravity in a program is quite different from gravity in real life due to the limitations of the programming environment. For example, programming gravity involves defining the acceleration due to gravity as a constant value, whereas in real life, the acceleration due to gravity varies based on factors such as altitude and latitude. Similarly, programming gravity involves using simplified models for air resistance and collisions between objects, which may not fully capture the complex interactions that occur in real-life situations. For my surface gravity, I kept gravity as a constant value at any distance from the surface. But, for my space gravity, I followed Newton's law of gravitational attraction which means that the object's acceleration varies depending on its distance from the object it is attracted to.

Another aspect of programming gravity is using discrete time steps to simulate the motion of objects, whereas in real life, the motion of objects is continuous. This may lead to inaccuracies in the simulation, particularly when objects are moving very quickly or are very small. However, for the sake of simplicity this is fine for me. In the future I would love to spend more time working on creating a small as possible time step, which may be achieved using the Runge-Kutta method.

Furthermore, programming gravity requires making decisions about how to represent gravity in a program. For example, I might need to decide how to handle collisions between objects that are affected by gravity. This decision will be influenced by the physics principles that govern gravity, as well as the specific requirements of the program. To make my simulation closer to real-life, I added in elastic collisions to my program. When two objects collide, a collision calculation is performed to determine the resulting velocities and angles at which the two objects will bounce off each other. In the future I would like to augment the program so that when two objects collide with each other, it would instead merge and create a bigger object. I've seen it done in different gravity simulations and it would be interesting to find how to implement that in my own program.

Since programming gravity is the main aspect of my senior project, I wanted to focus my efforts on creating a good simulation. However, I would still need to make a user interface that is able to interact with my program. Creating a UI takes time if done from scratch, time that I did not have. Luckily, there are many different resources online that provide pre-built and open-source frameworks.

Using Angular to make a web page provides several advantages over programming from scratch. The Angular framework is built around the concept of components, which are modular and reusable building blocks for creating web pages. These components can be easily reused across different pages, reducing code duplication, and making it easier to maintain and update the web page.

One of the benefits of using Angular is increased productivity. Angular provides a rich set of tools and features that can help developers be more productive. For example, the Angular Command Line Interface provides a set of commands for generating new components, services, and modules, which can save time and reduce the risk of errors. In addition, Angular can improve the performance of the web page. Angular uses a technique called Ahead-of-Time (AOT) compilation to convert templates into optimized JavaScript code. AOT compilation can reduce the size of the JavaScript code that needs to be downloaded by the browser, resulting in faster load times and better overall performance.

Using Angular allowed me to create a simple interface without spending time writing out stylesheets and functionality using JavaScript. Instead, I was able to whip up a simple and sleek interface using the pre-built classes and stylesheets and easily connect them to my gravity simulation code. In conclusion, using Angular and HTML5 Canvas to create a gravity simulation was a challenging but rewarding experience. The process involved extensive research into physics principles to better understand the workings of gravity and create a simulation that accurately reflected its behavior.

The simulation required a deep understanding of mathematical concepts, particularly calculus and differential equations, to accurately model the gravitational interactions between objects. It was important to carefully consider the laws of motion, including Newton's three laws, to create a simulation that behaved realistically.

In the future, I would be interested in creating a gravity simulation in a non-web application, or possibly using a game development platform such as Unity to replicate the simulation. I am also thinking of improving the current simulation by changing it from 2D, to 3D. Working in a three-dimensional environment does pose its own challenges, and it would require better graphics for a more visual appeal than 2D graphics. Other than the graphics, you are simply adding a z-value to the mix of calculations so it shouldn't be too difficult. However, if I've learned anything about programming after doing it for the past 4 years, it is that it will probably not be easy at all.

Despite the limitations of programming environments, I was able to simulate gravity in a program using mathematical equations and concepts that I had spent a hefty amount of time researching to fully grasp the concept. Making decisions about how to represent gravity in a program, such as handling collisions between objects, requires consideration of both physics' principles and program requirements. To make the user interface for the program, using pre-built and open-source frameworks like Angular can save time and increase productivity. In all, programming gravity is a challenging but rewarding task that requires both knowledge of physics and programming skills.

Chapter 6: Conclusion

It is safe to say that I achieved my main goal with this project, to create a gravity simulation in a web application that can replicate accurately gravity on both the surface of a planet and in space. After much research and studying the various physics and computer science concepts required to make this program, I was able to create a program that allows users to interact with the simulation and play around with the various parameters that affect the objects in the simulation.

The use of Angular and HTML5 Canvas provided a powerful framework for creating an interactive simulation that could be easily manipulated by users. The Canvas API made it easy to render realistic graphics and animations that accurately reflected the movements of objects under the influence of gravity.

One of the biggest challenges in creating the simulation was optimizing performance. The simulation involved many calculations and updates on every frame, which could easily overwhelm the browser if not handled correctly. Careful attention was paid to optimizing the code and reducing unnecessary calculations to ensure a smooth and efficient performance in the simulation.

Despite the challenges, the experience of exploring the world of computational physics was incredibly rewarding. It provided a deep appreciation for the complexity of the universe and the incredible power of mathematical models to help us better understand it. The simulation was not only a fun and engaging project, but it also had educational value, helping to teach users about the behavior of gravity and its impact on objects in the universe.

In closing, the use of Angular and HTML5 Canvas to create a gravity simulation was an exciting and challenging experience. It required a deep understanding of physics principles and mathematical concepts, as well as careful attention to performance optimization. However, the result was a realistic and engaging simulation that provided both entertainment and education for users. The project provided a glimpse into the fascinating world of computational physics and was a rewarding experience that will be remembered for a long time to come.

References

1. “OpenStax | Free Textbooks Online with No Catch.” *@Openstax/Os-webview*, openstax.org/details/books/university-physics-volume-1. 2022.
2. Shiffman, Daniel. “The Nature of Code”. *https://natureofcode.com/book/,* 2012.
3. “What Is User Interface (UI) Design?” *The Interaction Design Foundation*, https://www.interaction-design.org/literature/topics/ui-design.
4. Pop, Sanda Andreea. “Essential UI Design Tips for Creating a Good User Interface.” *TeleportHQ*, TeleportHQ, 2022, https://teleporthq.io/blog/design-tips-for-creating-a-good-user-interface.
5. “Angular Documentation.” Angular, https://angular.io/guide/developer-guide-overview.
6. "Introduction to Computational Physics" Richard Fitzpatrick, Richard Fitzpatrick. 2013.
7. "Gravity Simulation with JavaScript" Adam Pritchard. O'Reilly Media. Date 2012.
8. "Physics for JavaScript Games, Animation, and Simulations: With HTML5 Canvas" Adrian Dobre and Dev Ramtal. Apress. 2014.
9. "Physics for Game Developers" David M. Bourg. O'Reilly Media. 2001.
10. "Mathematics for Physics and Physicists" Walter Appel. Springer. 2014.
11. "The Elegant Universe: Superstrings, Hidden Dimensions, and the Quest for the Ultimate Theory" Brian Greene. W. W. Norton & Company. 2000.
12. "The Gravitational Universe: A Beginner's Guide" Nathalie Deruelle and Jean-Philippe Uzan. Oxford University Press. 2014.

Appendix (code)

Project can be found on https://github.com/NovaOrion/GravitySimulation.git

App.component.html

```
<mat-toolbar color="primary">
    <span>Maria's Gravity Sim - {{mode === AppMode.SurfaceGravity ? 'Surface
Gravity' : mode === AppMode.SpaceGravity ? 'Space Gravity' : 'About'}}</span>
    <span class="spacer"></span>
    <button mat-icon-button matTooltip="Surface View"
(click)="toggleMode(AppMode.SurfaceGravity)">
        <i class="fa-solid fa-earth-americas"></i>
    </button>
    <button mat-icon-button matTooltip="Space View"
(click)="toggleMode(AppMode.SpaceGravity)">
        <i class="fa-solid fa-meteor"></i>
    </button>
    <button mat-icon-button matTooltip="About"
(click)="toggleMode(AppMode.About)">
        <i class="fa-regular fa-circle-question"></i>
    </button>
</mat-toolbar>
<mat-drawer-container class="content" autosize *ngIf="mode!==AppMode.About"
[ngSwitch]="mode">
    <mat-drawer #drawer class="sidenav" mode="side" [opened]="isSideNavOpen"
*ngSwitchCase="AppMode.SurfaceGravity">
        <p style="padding-top: 8px">Parameters</p>
        <div class="param-field">
            <label for="fps" matTooltip="Animation Frames Per Second">FPS
<span class="label-dim">[{{fps}}]</span></label>
            <mat-slider id="fps" showTickMarks discrete min="30" max="90"
step="30" class="val">
                <input matSliderThumb [(ngModel)]="fps" >
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="trail" matTooltip="Number of trailing dots following a
scene object">Trail Length <span class="label-dim">[{{trail}}]</span></label>
            <mat-slider id="trail" showTickMarks discrete min="10" max="100"
step="10" class="val">
                <input matSliderThumb [ngModel]="trail"
(ngModelChange)="onTrailLengthChanged($event)">
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="num_of_balls" matTooltip="Number of ball for
simulation"># Balls <span class="label-dim">[{{num_of_balls}}]</span></label>
            <mat-slider id="num_of_balls" showTickMarks discrete min="1"
max="10" step="1" class="val">
```

```
                <input matSliderThumb [(ngModel)]="num_of_balls">
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="gravity" matTooltip="Gravity strength">Gravity <span
class="label-dim">[{{gravity}}]</span></label>
            <mat-slider id="gravity" showTickMarks discrete min="0" max="0.5"
step="0.05" class="val">
                <input matSliderThumb [ngModel]="gravity"
(ngModelChange)="onGravityChanged($event)">
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="elasticity" matTooltip="Energy preseved after bouncing
of the ground">Elasticity <span class="label-
dim">[{{elasticity}}]</span></label>
            <mat-slider id="elasticity" showTickMarks discrete min="0" max="1"
step="0.1" class="val">
                <input matSliderThumb [ngModel]="elasticity"
(ngModelChange)="onElasticityChanged($event)">
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="friction" matTooltip="Horizontal friction force
strength">Friction <span class="label-dim">[{{friction}}]</span></label>
            <mat-slider id="friction" showTickMarks discrete min="0"
max="0.06" step="0.02" class="val">
                <input matSliderThumb [ngModel]="friction"
(ngModelChange)="onFrictionChanged($event)">
            </mat-slider>
        </div>
    </mat-drawer>

    <mat-drawer #drawer class="sidenav" mode="side" [opened]="isSideNavOpen"
*ngSwitchCase="AppMode.SpaceGravity">
        <p style="padding-top: 8px">Parameters</p>
        <div class="param-field">
            <label for="fps">FPS <span class="label-
dim">[{{fps}}]</span></label>
            <mat-slider id="fps" showTickMarks discrete min="30" max="90"
step="30" class="val">
                <input matSliderThumb [(ngModel)]="fps" >
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="mass">Sun Mass <span class="label-
dim">[{{mass}}]</span></label>
```

```
            <mat-slider id="mass" showTickMarks discrete min="1" max="10"
step="1" class="val">
                <input matSliderThumb [ngModel]="mass"
(ngModelChange)="onSunMassChanged($event)" >
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="ball_mass">Planet Mass <span class="label-
dim">[{{ball_mass}}]</span></label>
            <mat-slider id="ball_mass" showTickMarks discrete min="1" max="10"
step="1" class="val">
                <input matSliderThumb [ngModel]="ball_mass"
(ngModelChange)="onBallMassChanged($event)" >
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="trail">Trail Length <span class="label-
dim">[{{pad_label(trail+'', 3)}}]</span></label>
            <mat-slider id="trail" showTickMarks discrete min="10" max="500"
step="10" class="val">
                <input matSliderThumb [ngModel]="trail"
(ngModelChange)="onTrailLengthChanged($event)">
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="num_of_balls"># Balls <span class="label-
dim">[{{num_of_balls}}]</span></label>
            <mat-slider id="num_of_balls" showTickMarks discrete min="0"
max="25" step="1" class="val">
                <input matSliderThumb [(ngModel)]="num_of_balls">
            </mat-slider>
        </div>
        <div class="param-field">
            <label for="gravity">Gravity <span class="label-
dim">[{{gravity}}]</span></label>
            <mat-slider id="gravity" showTickMarks discrete min="0" max="1"
step="0.1" class="val">
                <input matSliderThumb [ngModel]="gravity"
(ngModelChange)="onGravityChanged($event)">
            </mat-slider>
        </div>
        <div class="param-field checkbox">
            <label for="lockSun" matTooltip="Lock Sun in the center">Lock
Sun</label>
            <mat-slide-toggle [(ngModel)]="lockSun"></mat-slide-toggle>
        </div>
        <div class="param-field checkbox">
```

```
            <label for="showVelocityVector" matTooltip="Show velocity
vector">Show Velocity Vectors</label>
            <mat-slide-toggle [ngModel]="showVelocityVector"
(ngModelChange)="onShowVelocityVectorChanged($event)"></mat-slide-toggle>
        </div>
        <div class="param-field checkbox">
            <label for="showAccelerationVector" matTooltip="Show acceleration
vector">Show Acceleration Vectors</label>
            <mat-slide-toggle [ngModel]="showAccelerationVector"
(ngModelChange)="onShowAccelerationVectorChanged($event)"></mat-slide-toggle>
        </div>
    </mat-drawer>

    <div class="sidenav-content" (resized)="resize()">
        <canvas #stage id="stage"></canvas>
    </div>

</mat-drawer-container>
<div class="footer" *ngIf="mode!==AppMode.About;else elseBlock;">
    <label style="padding-right: 1em; color: white">Sounds</label>
    <mat-slide-toggle [ngModel]="playSound"
(ngModelChange)="togglePlaySound($event)" [title]="playSound?'Mute
sounds':'Play sounds'" style="margin-right: 2em"></mat-slide-toggle>
    <button mat-raised-button color="primary" class="footer-button"
[disabled]="symState!==SymState.Stopped" (click)="start()">
        Start Simulator
    </button>
    <button mat-raised-button color="primary" class="footer-button"
*ngIf="symState===SymState.Playing" (click)="pause()">
        Pause Simulator
     </button>
     <button mat-raised-button color="primary" class="footer-button"
*ngIf="symState===SymState.Paused" (click)="resume()">
        Resume Simulator
     </button>
    <button mat-raised-button color="primary" class="footer-button"
[disabled]="symState===SymState.Stopped" (click)="stop()">
        Stop Simulator
     </button>
    <mat-slide-toggle [(ngModel)]="isSideNavOpen"
[title]="isSideNavOpen?'Toggle to close Side Panel':'Toggle to open Side
Panel'"></mat-slide-toggle>
</div>
<ng-template #elseBlock>
    <div id="title">
        <h1>About This Project</h1>
    </div>
    <div id="aboutMyself">
```

```
        <div style="text-align: center; width: 100%"><img
src="assets/galaxy.jpg"/></div>
        <p>This research project aims to study key concepts of computational
physics by creating a simulation of gravity, with a focus on its
implementation and design of the simulation itself. The project involves
building up a solid knowledge of physics and differential equations that are
required for accurate simulations and exploring the limits of simulations
compared to real-life experimentation. It takes the concepts of physics and
programming and demonstrates their intersection, showing how physics
simulations are important for scientists to better visualize and understand
the world around us. By making a fun yet educational web application that can
be used by a variety of people, I want to emphasize the importance of a well-
designed user interface in creating a positive user experience. My ultimate
goal is to demonstrate my expertise and capabilities in creating a complex
program through this project.</p>
    </div>
    <div id="links-box">
        <div>
            <a href="https://github.com/NovaOrion/gravity-sim.git"
target="_blank" class="links">Link to Code on GitHub</a>
        </div>
        <div>
            <a href="/src/SeniorThesisDraft.pdf" download
class="links">Download PDF</a>
        </div>
    </div>
    <div style="text-align: center; margin-top: 2em">
        <div style="display: block; font-weight: bold" >Let's start</div>
        <div style="margin-top: 1em;">
            <button (click)="toggleMode(AppMode.SurfaceGravity)"
style="margin-right: 1em">Surface Graviry Sym</button>
            <button (click)="toggleMode(AppMode.SpaceGravity)">Space Gravity
Sym</button>
        </div>
    </div>
</ng-template>
<audio id="hit1">
    <source src="assets/hit1.mp3" type="audio/mpeg" />
</audio>
<audio id="hit2">
    <source src="assets/hit2.mp3" type="audio/mpeg" />
</audio>
<audio id="hit3">
    <source src="assets/hit3.mp3" type="audio/mpeg" />
</audio>
```

App.component.scss

```scss
:host {
    height: 100%;
    display: flex;
    flex-direction: column;
}
h1 {
    font-size: 45px;
  }
#title {
    display: flex;
    justify-content: center;
    padding-top: 30px;
}
#aboutMyself {
    padding-left: 70px;
    padding-right: 70px;

    justify-content: center;
    text-align: justify;
    p {
        text-indent: 2em;
        &::first-letter {
            font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida
Grande', 'Lucida Sans', Arial, sans-serif;
            font-weight: bold;
            font-size: 20px;
            color: darkblue;
        }
    }
    img {
        margin: 1em 0;
        border-radius: 13px;
    }
}
#links-box {
    display: flex;
    justify-content: center;
}
.links {
    font-size: medium;
    padding: 10px;
}
```

App.component.ts

```
import { AfterViewInit, Component, ElementRef, NgZone, OnDestroy, OnInit,
ViewChild } from '@angular/core';
import * as _ from 'lodash';
import { AppMode, hitTestCircle, ICircle, IScene, ISceneObject, IVector,
SymState, uuid } from 'src/common/common';
import { Vector } from 'src/common/vector';
import { Ball, IBallOptions } from 'src/simulator/ball';
import { Scene } from 'src/simulator/scene';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements AfterViewInit, OnDestroy {
  @ViewChild('stage') canvas!: ElementRef<HTMLCanvasElement>;
  public ctx: CanvasRenderingContext2D | null = null;
  public world = 1000;
  public title = 'Maria Gravity Project';
  public mass = 2;
  public ball_mass = 2;
  public trail = 100;
  public fps = 60;
  public isSideNavOpen=true;
  public timeoutId: any;
  private requestId: number = 0;
  public symState: SymState = SymState.Stopped;
  public scene: IScene | null = null;
  public borderSize: number = 2;
  public num_of_balls: number = 0;
  public gravity: number = 0.4;
  public elasticity: number = 0.9;
  public friction: number = 0.06;
  public mode: AppMode = AppMode.About;
  public AppMode = AppMode;
  public SymState = SymState;
  public lockSun: boolean = true;
  public start_drawing: IVector | null = null;
  public showVelocityVector: boolean = false;
  public showAccelerationVector: boolean = false;
  public playSound: boolean = false;


  /**
   * Constructor for main app component
   * @param zone NG zone
   */
```

```
  constructor(private zone: NgZone) {
  }

  public togglePlaySound(playSound: boolean) {
    this.playSound = playSound;
    if (this.scene) {
      this.scene.playSound = this.playSound;
    }
  }

  public toggleMode(mode: AppMode): void {
    this.mode = mode;
    this.num_of_balls = AppMode.SurfaceGravity ? 10 : 0;
    this.trail = AppMode.SurfaceGravity ? 100 : 100;
    this.stop();
    _.delay(() => this.initSym(), 100);
  }

  private canStartHere(c2: ICircle): boolean {
    if (!this.scene) return false;
    let result = this.scene.objects.some(x => {
      if (x instanceof Ball) {
        const c1 = x as ICircle;
        return hitTestCircle(c1, c2);
      }
      return false;
    });
    return !result;
  }


  public addRandomBall(scene: IScene, options?: any): void {
    if (!this.scene) return;
    const colors = ["greed", "red", "yellow", "#AED6F1", "white", "#F5CBA7",
"pink", "orange", "cyan"];

    let r = 3 + Math.random() * (this.mode === AppMode.SurfaceGravity ? 10 :
5);
    let center;
    if (options.center) {
      center = options.center;
      let c: ICircle = {
        center,
        radius: r
      };
      if (!this.canStartHere(c)) {
        return;
      }
    } else {
```

```
      center = { x: r + Math.random() * (this.world - 2 * r) , y: r +
Math.random() * (this.scene.VisibleWorldHeight - 2 * r) };
      let c: ICircle = {
        center,
        radius: r
      };
      while (!this.canStartHere(c)) {
        r = 5 + Math.random() * 10;
        center = { x: r + Math.random() * (this.world - 2 * r) , y: r +
Math.random() * (this.scene.VisibleWorldHeight - 2 * r) };
        c = {
          center,
          radius: r
        };
      }
    }

    const name = _.get(options, "name", `ball-${uuid()}`);
    const ball = new Ball(name, {
      center,
      radius: r,
      color: _.get(options, "color", colors[Math.ceil(Math.random() *
colors.length)]),
      speed: _.get(options, "speed", Math.random() * 5),
      angle: _.get(options, "angle", Math.random() * 360),
      mass: _.get(options, "mass", r * 10),
      trace: true,
      trace_limit: this.trail
    });
    this.scene?.add(ball);
  }

  public addSun(scene: IScene) {
    if (this.mode === AppMode.SpaceGravity) {
      const ball = new Ball("sun", {
        center: {x: this.world / 2, y: scene.VisibleWorldHeight / 2},
        radius: 15,
        color: "orange",
        speed: 0,
        angle: 0,
        mass: this.mass * 1000,
        trace: true,
        trace_limit: this.trail
      });
      this.scene?.add(ball);
    }
  }
```

```
/**
 * Start Simulation
 */
public start(): void {
  if (!this.scene) return;

  for (let i = 0; i < this.num_of_balls; ++i) {
    this.addRandomBall(this.scene, { name: `ball-${i}`});
  }

  this.addSun(this.scene);

  this.symState = SymState.Playing;
}

/**
 * Pause Simulation
 */
public pause(): void {
  this.symState = SymState.Paused;
  this.scene!.inPause = true;
}

/**
 * Resume Simulation
 */
public resume(): void {
  this.symState = SymState.Playing;
  this.scene!.inPause = false;
}

public pad_label(str: string, num: number) {
  return _.padStart(str, num, ' ');
}

/**
 * Stop Simulation
 */
public stop(): void {
  if (this.scene) {
    this.scene?.clear();
    this.scene!.inPause = false;
  }
  this.symState = SymState.Stopped;
}

public onTrailLengthChanged(trail_length: number): void {
  this.trail = trail_length;
```

```
    this.scene?.updateWithCondition(x => true, x => {
      x.trace_limit = trail_length;
      return x;
    });
  }
  public onGravityChanged(gravity: number): void {
    this.scene!.gravity = this.gravity = gravity;
  }
  public onElasticityChanged(e: number): void {
    this.scene!.elasticity = this.elasticity = e;
  }
  public onFrictionChanged(f: number): void {
    this.scene!.friction = this.friction = f;
  }
  public onSunMassChanged(m: number): void {
    this.mass = m;
    const sun = this.scene?.objects.get("sun") as Ball;
    sun.mass = m * 1000;
    sun.radius = sun.mass > 5001 ? 25 : 15;
    this.scene?.updateByKey("sun", sun);
  }
  public onBallMassChanged(m: number): void {
    this.ball_mass = m;
    this.scene?.updateWithCondition(x => x.name !== 'sun', x => {
      if (x instanceof Ball) {
        (x as Ball).mass = this.ball_mass * 10;
      }
      return x;
    });
  }
  public onShowVelocityVectorChanged(show: boolean) {
    this.scene!.showVelocityVector = show;
  }
  public onShowAccelerationVectorChanged(show: boolean) {
    this.scene!.showAccelerationVector = show;
  }

  public initSym(): void {
    if (this.mode === AppMode.About) {
      return;
    }

    this.ctx = this.canvas.nativeElement.getContext("2d");
    this.setCanvasSize();

    // Scene initialization
    this.scene = new Scene(this.ctx!, this.world, this.borderSize);
    this.scene.mode = this.mode;
```

```
    this.scene.gravity = this.gravity;
    this.scene.elasticity = this.elasticity;
    this.scene.friction = this.friction;
    this.scene.playSound = this.playSound;

    this.scene.postCalc = () => {
      if (this.mode === AppMode.SpaceGravity && this.lockSun) {
        const sun = this.scene?.objects.get('sun');
        if (sun && this.scene) {
          const sunVector = new Vector(sun.position.x, sun.position.y);
          const centerVector = new Vector(this.world / 2,
this.scene.VisibleWorldHeight / 2);
          const pan = centerVector.sub(sunVector);
          if (pan.magnitude > 0.001) {
            this.scene.updateWithCondition(x => true, x => {
              x.pan(pan);
              return x;
            });
          }
        }
      }
    }

    this.canvas.nativeElement.addEventListener("mousedown", e => {
      if (this.symState !== SymState.Playing) return;
      this.start_drawing = new Vector(e.clientX, e.clientY);
    });
    this.canvas.nativeElement.addEventListener("mouseup", e => {
      if (this.symState !== SymState.Playing || this.start_drawing === null)
return;
      const end_drawing = new Vector(e.clientX, e.clientY);
      const angle = end_drawing.sub(this.start_drawing!).angle();
      const rect = this.canvas.nativeElement.getBoundingClientRect();
      const pos = {
        x: this.start_drawing!.x - rect.left,
        y: this.start_drawing!.y - rect.top
      };
      this.addRandomBall(this.scene!, {
        center: {
          x: this.scene!.worldX(pos.x),
          y: this.scene!.worldY(pos.y)
        },
        angle: -1 * angle * 180 / Math.PI, // into degrees,
      });

      this.start_drawing = null;
    });
```

```
    this.zone.runOutsideAngular(() =>
      this.animate()
    );
  }

  /**
   * The DOM is loaded and all bindings are available
   */
  ngAfterViewInit(): void {
    this.initSym();
  }

  /**
   * Clear canvas
   */
  public clear(): void {
    if (!this.ctx || !this.canvas || !this.canvas.nativeElement) return;
    this.ctx.clearRect(0, 0, this.canvas.nativeElement.width,
this.canvas.nativeElement.height);
  }

  /**
   * Draw background and all scene objects
   * Internally, every object calls it's delta method before drawing
   */
  public draw(): void {
    if (!this.ctx) return;
    this.ctx.beginPath();
    this.ctx.lineWidth = this.borderSize;
    this.ctx.rect(0, 0, this.canvas.nativeElement.width,
this.canvas.nativeElement.height);
    this.ctx.fillStyle = "black";
    this.ctx.fill();
    this.ctx.strokeStyle = "white";
    this.ctx.stroke();
    this.scene?.draw();
  }

  /**
   * Resize canvas and scene
   */
  public resize(): void {
    this.setCanvasSize();
    this.scene?.resize();
    this.draw();
  }

  /**
```

```
   * Setting Canvas size based on parent element
   */
  protected setCanvasSize(): void {
    if (!this.canvas) return;
    const parent: HTMLElement | null =
this.canvas.nativeElement.parentElement;
    if (!parent) return;
    var rect = parent.getBoundingClientRect();
    this.canvas.nativeElement.width = rect.width;
    this.canvas.nativeElement.height = rect.height;
  }

  /**
   * Animation loop
   */
  protected animate(): void {
    this.clear();
    this.draw();
    this.requestId = requestAnimationFrame(() => {
      this.timeoutId = _.delay(() => this.animate(), 1000/this.fps);
    });
  }

  /**
   * Called on closing web application
   */
  public ngOnDestroy(): void {
    clearTimeout(this.timeoutId);
    cancelAnimationFrame(this.requestId);
  }

}
```

Common.ts

```
import {OrderedMap} from 'immutable';
import * as _ from 'lodash';

export interface ISceneObject {
    name: string;
    enabled: boolean;
    position: IPoint;
    trace?: boolean;
    trace_limit?: number;
    delta(scene: IScene): void;
    collide(scene: IScene): void;
    draw(scene: IScene): void;
    draw_trace(scene: IScene): void;
    pan(p: IVector): void;
}
export interface IScene {
    ctx: CanvasRenderingContext2D;
    objects: OrderedMap<string, ISceneObject>;
    add(obj: ISceneObject): IScene;
    remove(name: string): IScene;
    hide(name: string): IScene;
    show(name: string): IScene;
    hideAllButOne(name: string): IScene;
    updateByKey(name: string, obj: ISceneObject): IScene;
    update(obj: ISceneObject | ISceneObject[]): IScene;
    updateWithCondition(condition: (obj: ISceneObject) => boolean, map: (obj:
ISceneObject) => ISceneObject): void;
    clear(): IScene;
    postCalc: (() => void) | null;
    draw(): void;
    resize(): void;
    X(x: number): number;
    Y(y: number): number;
    worldX(x: number): number;
    worldY(y: number): number;
    scale: number;
    VisibleWorldHeight: number;
    width: number;
    height: number;
    gravity: number;
    elasticity: number;
    friction: number;
    mode: AppMode;
    inPause: boolean;
    showVelocityVector: boolean;
    showAccelerationVector: boolean;
    playSound: boolean;
```

```
}
export interface IPoint {
    x: number;
    y: number;
}
export interface IRect {
    x: number;
    y: number;
    width: number;
    height: number;
}
export interface ISize {
    width: number;
    height: number;
}
export interface IVector {
    x: number;
    y: number;
    add(v: IVector) : IVector;
    sub(v: IVector): IVector;
    mul(n: number): IVector;
    div(n: number): IVector;
    magnitude: number;
    normalize(): IVector;
    dir(direction: number): IVector;
    dotProduct(v: IVector): number;
    crossProduct(v: IVector): number;
    toString(): string;
    isNull: boolean;
    angle(): number;
}
export interface ICircle {
    center: IPoint;
    radius: number;
}
export function hitTestCircle(c1: ICircle, c2: ICircle): boolean {
    let result = false;
    const dx = c1.center.x - c2.center.x;
    const dy = c1.center.y - c2.center.y;
    const distance = (dx*dx + dy*dy);
    if (distance <= (c1.radius + c2.radius) * (c1.radius + c2.radius)) {
        result = true;
    }
    return result;
}

export enum AppMode {
    About,
```

```
    SurfaceGravity,
    SpaceGravity
}

export enum SymState {
    Playing,
    Paused,
    Stopped
}

export function uuid() {
    let uuidValue = "", k, randomValue;
    for (let k = 0; k < 32; k++) {
        randomValue = Math.random() * 16 | 0;

        if (k == 8 || k == 12 || k == 16 || k == 20) {
            uuidValue += "-";
        }
        uuidValue += (k == 12 ? 4 : (k == 16 ? (randomValue & 3 | 8) :
randomValue)).toString(16);
    }
    return uuidValue;
}

export function drawArrowhead(ctx: CanvasRenderingContext2D, from: IPoint, to:
IPoint, radius: number, color: string) {
    var x_center = to.x;
    var y_center = to.y;

    var angle;
    var x;
    var y;

    ctx.beginPath();
    angle = Math.atan2(to.y - from.y, to.x - from.x)
    x = radius * Math.cos(angle) + x_center;
    y = radius * Math.sin(angle) + y_center;
    ctx.moveTo(x, y);
    angle += (1.0/3.0) * (2 * Math.PI)
    x = radius * Math.cos(angle) + x_center;
    y = radius * Math.sin(angle) + y_center;
    ctx.lineTo(x, y);
    angle += (1.0/3.0) * (2 * Math.PI)
    x = radius * Math.cos(angle) + x_center;
    y = radius * Math.sin(angle) + y_center;
    ctx.lineTo(x, y);
    ctx.closePath();
    ctx.fillStyle = color;
```

```
    ctx.fill();
}
```

Vector.ts

```
import { IPoint, IVector } from "./common";

export class Vector implements IVector {

    constructor(public x: number, public y: number) {
    }
    public add(v: IVector) : IVector {
        return new Vector(this.x + v.x, this.y + v.y);
    }
    public sub(v: IVector): IVector {
        return new Vector(this.x - v.x, this.y - v.y);
    }
    public mul(n: number): IVector {
        return new Vector(this.x * n, this.y * n);
    }
    public div(n: number): IVector {
        return new Vector(this.x / n, this.y / n);
    }
    static move(pt1: IPoint, pt2: IPoint): IVector {
        const x = pt2.x - pt1.x;
        const y = pt2.y - pt1.y;
        return new Vector(x, y);
    }
    public get isNull(): boolean {
        return this.x === 0 && this.y === 0;
    }
    public get magnitude(): number {
        return Math.sqrt(this.x*this.x + this.y*this.y);
    }
    public normalize(): IVector {
        const magnitude = this.magnitude;
        return this.div(magnitude);
    }
    public dir(direction: number): IVector {
        const magnitude = this.magnitude;
        return new Vector(Math.cos(direction) * magnitude, Math.sin(direction)
* magnitude);
    }
    public angle(): number {
        return Math.atan2(this.y, this.x);
    }
    public dotProduct(v: IVector): number {
```

```
        return (this.x * v.x) + (this.y * v.y);
    }
    public crossProduct(v: IVector): number {
        return (this.x * v.x) - (this.y - v.y);
    }

    // static methods
    static unitVector(direction: number): IVector {
        return new Vector(Math.cos(direction), Math.sin(direction));
    }
    static fromPolar(length: number, angle: number): IVector {
        return new Vector(length * Math.cos(angle), length * Math.sin(angle));
    }

    public toString(): string {
        return `Vector <${this.x},${this.y}>`;
    }
}
```

Ball.ts

```
import { AppMode, drawArrowhead, hitTestCircle, ICircle, IPoint, IScene,
IVector } from "src/common/common";
import { BaseObject } from "./base";
import * as _ from "lodash";
import { Vector } from "src/common/vector";

export interface IBallOptions {
    center: IPoint;
    radius: number;
    color: string;
    speed: number;
    angle: number;
    trace?: boolean;
    trace_limit?: number;
    mass: number;
}
export class Ball extends BaseObject implements ICircle {

    public radius: number;
    public color: string;
    public speed: number;
    public angle: number;
    public mass: number;


    private radians: number = 0;
```

```
    private x_velocity: number = 0;
    private y_velocity: number = 0;
    private acceleration_vector: IVector | null = null;

    protected wallHitSound: HTMLAudioElement | null = null;
    protected ballHitSound: HTMLAudioElement | null = null;
    protected sunHitSound: HTMLAudioElement | null = null;

    constructor(name: string, public options: IBallOptions) {
        super(name);
        this.position = this.options.center;
        this.radius = this.options.radius;
        this.color = this.options.color;
        this.speed = this.options.speed;
        this.angle = this.options.angle;
        this.mass = this.options.mass;
        this.trace = _.get(this.options, "trace", false);
        this.trace_limit = _.get(this.options, "trace_limit", 0);
        this.bounds = { width: 2 * this.radius, height: 2 * this.radius };
        this.radians = this.angle * Math.PI / 180;
        this.x_velocity = Math.cos(this.radians) * this.speed;
        this.y_velocity = Math.sin(this.radians) * this.speed;
        this.update();
    }

    private update(scene?: IScene): void {
        if (scene && scene.gravity && scene.mode === AppMode.SurfaceGravity) {
            this.y_velocity -= scene.gravity;
        }

        if (scene && scene.mode === AppMode.SpaceGravity) {
            this.acceleration_vector = scene.objects.reduce((acc, ball) => {
                if (this.name === ball.name || !(ball instanceof Ball)) return
acc;

                let vec = Vector.move(
                    {x: this.position.x, y: this.position.y},
                    {x: ball.position.x, y: ball.position.y}
                );

                const strength = scene.gravity/10000 * this.mass * ball.mass /
vec.magnitude * vec.magnitude;
                vec = vec.normalize().mul(strength / this.mass); // this is F
/ m = a acceleration
                acc = acc.isNull ? vec : acc.add(vec);
                return acc;
            }, new Vector(0, 0));
```

```
            this.x_velocity += this.acceleration_vector.x;
            this.y_velocity += this.acceleration_vector.y;
        }

        this.position.x += this.x_velocity;
        this.position.y += this.y_velocity;
    }

    private testWalls(scene: IScene): boolean {
        const bw = this.bounds.width / 2;
        const bh = this.bounds.height / 2;

        if (scene.X(this.position.x + bw) > scene.width) {
            this.x_velocity = -1 * this.x_velocity;
            this.position.x = scene.width / scene.scale - bw - 1;
            return true;
        } else if (scene.X(this.position.x - bw) < scene.X(0)) {
            this.x_velocity = -1 * this.x_velocity;
            this.position.x = bw + 1;
            return true;
        } else if (this.position.y > scene.VisibleWorldHeight) {
            this.y_velocity = -1 * this.y_velocity;
            this.position.y = scene.VisibleWorldHeight - bh - 1;
            return true;
        } else if (scene.Y(this.position.y - bh) >= scene.Y(0)) {
            if (scene && scene.gravity && scene.elasticity && scene.mode ===
AppMode.SurfaceGravity) {
                this.y_velocity = -1 * this.y_velocity * scene.elasticity;
            } else {
                this.y_velocity = -1 * this.y_velocity;
            }
            if (scene && scene.gravity && scene.friction && scene.mode ===
AppMode.SurfaceGravity) {
                this.x_velocity = this.x_velocity - (this.x_velocity *
scene.friction);
            }
            this.position.y = bh + ((scene.mode === AppMode.SpaceGravity) ? 1
: 0);
            return true;
        }
        return false;
    }

    private process_collide(scene: IScene, ball: Ball) {
        const ball2 = ball;

        const dx = this.position.x - ball2.position.x;
        const dy = this.position.y - ball2.position.y;
```

```
        const collisionAngle = Math.atan2(dy, dx);

        const speed1 = Math.sqrt(this.x_velocity * this.x_velocity +
this.y_velocity * this.y_velocity);
        const speed2 = Math.sqrt(ball2.x_velocity * ball2.x_velocity +
ball2.y_velocity * ball2.y_velocity);

        const direction1 = Math.atan2(this.y_velocity, this.x_velocity);
        const direction2 = Math.atan2(ball2.y_velocity, ball2.x_velocity);

        const x_velocity_1 = speed1 * Math.cos(direction1 - collisionAngle);
        const y_velocity_1 = speed1 * Math.sin(direction1 - collisionAngle);
        const x_velocity_2 = speed2 * Math.cos(direction2 - collisionAngle);
        const y_velocity_2 = speed2 * Math.sin(direction2 - collisionAngle);

        const final_x_velocity_1 = ((this.mass - ball2.mass) * x_velocity_1 +
(ball2.mass + ball2.mass) * x_velocity_2)/(this.mass + ball2.mass);
        const final_x_velocity_2 = ((this.mass + this.mass) * x_velocity_1 +
(ball2.mass - this.mass) * x_velocity_2)/(this.mass + ball2.mass);

        const final_y_velocity_1 = y_velocity_1;
        const final_y_velocity_2 = y_velocity_2;

        this.x_velocity = Math.cos(collisionAngle) * final_x_velocity_1 +
Math.cos(collisionAngle + Math.PI/2) * final_y_velocity_1;
        this.y_velocity = Math.sin(collisionAngle) * final_x_velocity_1 +
Math.sin(collisionAngle + Math.PI/2) * final_y_velocity_1;
        ball2.x_velocity = Math.cos(collisionAngle) * final_x_velocity_2 +
Math.cos(collisionAngle + Math.PI/2) * final_y_velocity_2;
        ball2.y_velocity = Math.sin(collisionAngle) * final_x_velocity_2 +
Math.sin(collisionAngle + Math.PI/2) * final_y_velocity_2;

        this.position.x += this.x_velocity;
        this.position.y += this.y_velocity;
        ball2.position.x += ball2.x_velocity;
        ball2.position.y += ball2.y_velocity;
        scene.updateByKey(this.name, this);
        scene.updateByKey(ball2.name, ball2);
    }

    override collide(scene: IScene): void {

        if (this.name === 'sun') return;
        const hit_object = scene.objects.find(x => {
            if (x instanceof Ball && x.name !== this.name) {
                const c1 = x as ICircle;
                return hitTestCircle(c1, this);
```

```
            }
            return false;
        });
        if (hit_object) {
            const b = hit_object as Ball;
            if (scene.mode === AppMode.SpaceGravity && b.name === 'sun')
{
                if (this.sunHitSound && scene.playSound) {
                    (this.sunHitSound.cloneNode(true) as
HTMLAudioElement).play();
                }
                scene.remove(this.name);
            } else {
                if (this.ballHitSound && scene.playSound) {
                    (this.ballHitSound.cloneNode(true) as
HTMLAudioElement).play();
                }
                this.process_collide(scene, b);
            }
        }
    }

    override delta(scene: IScene): void {
        super.delta(scene);
        this.update(scene);
        if (scene.mode === AppMode.SurfaceGravity) {
            const hitWall = this.testWalls(scene);
            if (hitWall && this.wallHitSound && this.y_velocity > 3 &&
scene.playSound) {
                (this.wallHitSound.cloneNode(true) as
HTMLAudioElement).play();
            }
        }
        scene.updateByKey(this.name, this);
    }

    override draw(scene: IScene): void {

        if (!this.wallHitSound) {
            this.wallHitSound = document.getElementById("hit1") as
HTMLAudioElement;
        }
        if (!this.ballHitSound) {
            this.ballHitSound = document.getElementById("hit2") as
HTMLAudioElement;
        }
        if (!this.sunHitSound) {
```

```
            this.sunHitSound = document.getElementById("hit3") as
HTMLAudioElement;
        }

        const ctx = scene.ctx;
        if (ctx && this.radius > 0) {
            const x = scene.X(this.position.x);
            const y = scene.Y(this.position.y);

            if (this.name !== 'sun' && scene.showVelocityVector) {
                const magnitude = Math.min(Math.max(this.radius * this.speed *
2, 10), 100);
                const vec = new Vector(this.x_velocity,
this.y_velocity).normalize().mul(magnitude);
                ctx.beginPath();
                ctx.moveTo(x, y);
                ctx.lineTo(scene.X(this.position.x + vec.x),
scene.Y(this.position.y + vec.y));
                ctx.lineWidth = 2;
                ctx.strokeStyle = "rgba(255, 0, 0, 0.5)";
                ctx.stroke();
                drawArrowhead(ctx, {x, y}, {x: scene.X(this.position.x +
vec.x), y: scene.Y(this.position.y + vec.y)}, 6, "rgba(255, 0, 0, 0.5)");
            }

            if (this.name !== 'sun' && scene.showAccelerationVector &&
this.acceleration_vector) {
                const magnitude =
Math.min(Math.max(this.acceleration_vector.magnitude * 200, 10), 100);
                const vec =
this.acceleration_vector.normalize().mul(magnitude);
                ctx.beginPath();
                ctx.moveTo(x, y);
                ctx.lineTo(scene.X(this.position.x + vec.x),
scene.Y(this.position.y + vec.y));
                ctx.lineWidth = 2;
                ctx.strokeStyle = "rgba(0, 255, 0, 0.5)";
                ctx.stroke();
                drawArrowhead(ctx, {x, y}, {x: scene.X(this.position.x +
vec.x), y: scene.Y(this.position.y + vec.y)}, 6, "rgba(0, 255, 0, 0.5)");
            }

            const radius = scene.scale * this.radius;
            ctx.beginPath();
            ctx.arc(x, y, radius, 0, 2 * Math.PI, false);
            if (this.name === 'sun') {
                const gradient = ctx.createRadialGradient(x, y, 0.1*radius, x,
y, radius);
```

```
                const sun_colors = [
                    ["red", "orange", "yellow"],
                    ["lightblue", "cyan", "white"]
                ];
                const i = this.mass > 5001 ? 1 : 0
                gradient.addColorStop(0, sun_colors[i][0]);
                gradient.addColorStop(0.8, sun_colors[i][1]);
                gradient.addColorStop(1, sun_colors[i][2]);
                ctx.shadowColor = this.mass > 5001 ? "white" : "yellow";
                ctx.shadowBlur = 25 + Math.round(Math.random() * 7);
                ctx.fillStyle = gradient;
            } else {
                ctx.shadowBlur = 0;
                ctx.fillStyle = this.color;
            }
            ctx.fill();
        }

    }

    public get center(): IPoint {
        return this.position;
    }
}
```

Base.ts

```
import { IPoint, IScene, ISceneObject, ISize, IVector } from
"src/common/common";

export abstract class BaseObject implements ISceneObject {

    public enabled: boolean = true;
    public trace: boolean = false;
    public trace_limit: number = 0;
    public position: IPoint;

    public bounds: ISize = { width: 0, height: 0 };
    public trace_points: IPoint[] = [];


    constructor(public name: string) {
        this.position = {x: 0, y: 0};
    }

    collide(scene: IScene): void {
```

```
    }

    pan(p: IVector): void {
        this.position.x += p.x;
        this.position.y += p.y;
        if (this.trace) {
            this.trace_points.forEach(tp => {
                tp.x += p.x;
                tp.y += p.y;
            });
        }
    }

    delta(scene: IScene): void {
        if (this.trace) {
            if (this.trace_limit > 0 && (this.trace_points.length + 1) >
this.trace_limit) {
                const to_remove = this.trace_points.length + 1 -
this.trace_limit;
                if (to_remove > 0) {
                    this.trace_points = this.trace_points.slice(to_remove);
                }
            }
            this.trace_points.push({x: this.position.x, y:
this.position.y});
        }
    }

    draw_trace(scene: IScene): void {
        const ctx = scene.ctx;
        if (this.trace && ctx) {
            this.trace_points.forEach((pt, index) => {
                let visible = true;
                if (scene.X(pt.x) > scene.width || scene.X(pt.x) < 0) {
                    visible = false;;
                }  else if (scene.Y(pt.y) > scene.height || scene.Y(pt.y) < 0)
{
                    visible = false;
                }
                if (!visible) return;
                ctx.beginPath();
                ctx.arc(scene.X(pt.x), scene.Y(pt.y), 1, 0, 2 * Math.PI);
                const alpha = this.trace_limit > 0 ?
index/(Math.min(this.trace_limit, this.trace_points.length)) : 1;
                ctx.fillStyle = `rgba(127, 127, 127, ${alpha})`;
                ctx.fill();
            });
        }
```

```
    }

    abstract draw(scene: IScene) : void;
}
```

Scene.ts

```
import {OrderedMap} from 'immutable';
import * as _ from 'lodash';
import { AppMode, IScene, ISceneObject } from 'src/common/common';

export class Scene implements IScene {

    protected sceneWidth: number = 0;
    protected sceneHeight: number = 0;
    protected padding: number = 0;
    public gravity: number = 0;
    public elasticity: number = 0;
    public friction: number = 0;
    public mode: AppMode = AppMode.SurfaceGravity;
    public inPause: boolean = false;
    public postCalc: (() => void) | null = null;
    public objects = OrderedMap<string, ISceneObject>();
    public showVelocityVector: boolean = false;
    public showAccelerationVector: boolean = false;
    public playSound: boolean = false;

    constructor(public ctx: CanvasRenderingContext2D, public world: number,
padding: number) {
        this.padding = padding;
        this.resize();
    }

    public resize(): void {
        this.sceneWidth = this.ctx?.canvas.width - 2 * this.padding;
        this.sceneHeight = this.ctx?.canvas.height - 2 * this.padding;
    }

    public get scale(): number {
        return this.sceneWidth / this.world;
    }

    public X(x: number): number {
        return Math.round(this.scale * x) + this.padding;
    }

    public Y(y: number): number {
```

```
        return this.ctx?.canvas.height - this.padding - Math.round(this.scale
* y);
    }

    public worldX(x: number): number {
        return (x - this.padding)/this.scale;
    }

    public worldY(y: number): number {
        return (this.ctx?.canvas.height - this.padding -
y)/this.scale;
    }

    public get VisibleWorldHeight(): number {
        return this.sceneHeight / this.scale;
    }

    // add to scene collection
    public add(obj: ISceneObject): IScene {
        this.objects = this.objects.set(obj.name, obj);
        return this;
    }

    // remove from scene collection
    public remove(name: string): IScene {
        this.objects = this.objects.delete(name);
        return this;
    }

    // hide object
    public hide(name: string): IScene {
        const obj = this.objects.get(name);
        if (obj) {
            obj.enabled = false;
            this.objects = this.objects.set(name, obj);
        }
        return this;
    }

    // show object
    public show(name: string): IScene {
        const obj = this.objects.get(name);
        if (obj) {
            obj.enabled = true;
            this.objects = this.objects.set(name, obj);
        }
        return this;
    }
```

```
    // hide all objects but one specified by name
    public hideAllButOne(name: string): IScene {
        this.objects.reduce((acc, val, key) => {
            val.enabled = (key === name);
            return acc.set(key, val);
        }, OrderedMap<string, ISceneObject>());
        return this;
    }

    // update object by key
    public updateByKey(name: string, obj: ISceneObject): IScene {
        this.objects = this.objects.set(name, obj);
        return this;
    }

    // update obj or array of objects
    public update(obj: ISceneObject | ISceneObject[]): IScene {
        const inp: ISceneObject[] = _.isArrayLike(obj) ? obj : [obj];
        for (let i=0; i<inp.length; ++i) {
            this.updateByKey(inp[i].name, inp[i]);
        }
        return this;
    }

    public updateWithCondition(condition: (obj: ISceneObject) => boolean, map:
(obj: ISceneObject) => ISceneObject): void {
        const filtered = this.objects.filter(condition).valueSeq().toArray();
        const remapped = filtered.map(map);
        this.update(remapped);
    }

    // remove all objects from scene
    public clear(): IScene {
        this.objects = this.objects.clear();
        return this;
    }

    // calculate delta and draw for each objects in scene in ordered manner
    public draw(): void {
        if (!this.inPause) {
            this.objects.forEach(x => x.delta(this));
            this.objects.forEach(x => x.collide(this));
            if (this.postCalc) {
                this.postCalc();
            }
        }
```

```
        const filtered = this.objects.filter(x => x.enabled);
        filtered.forEach(x => x.draw_trace(this));
        filtered.forEach(x => x.draw(this));
    }

    public get width(): number {
        return this.sceneWidth;
    }

    public get height(): number {
        return this.sceneHeight;
    }
}
```