# Asteroid Collision Simulator

## Introduction

The Asteroid Collision Simulator project addresses the challenge of accurately predicting collisions in an asteroid field over time. In space exploration and astronomy, understanding when and where asteroids might collide is crucial for both scientific research and potential space mission safety. This simulation tracks multiple asteroids with different sizes and velocities moving in a two-dimensional plane, detecting all points of intersection that represent potential collisions.

## Objectives

1. **Accurate Collision Detection**: Identify all instances where two or more asteroids occupy the same space.
2. **Time-Based Simulation**: Track asteroid movements over specified time intervals.
3. **Visual Representation**: Create an animation of asteroid movements and collision events.
4. **Efficiency**: Process large numbers of asteroids with reasonable computational resources.
5. **Clear Reporting**: Generate readable output files documenting all collision events.

## Methodology

### Collision Detection Algorithm

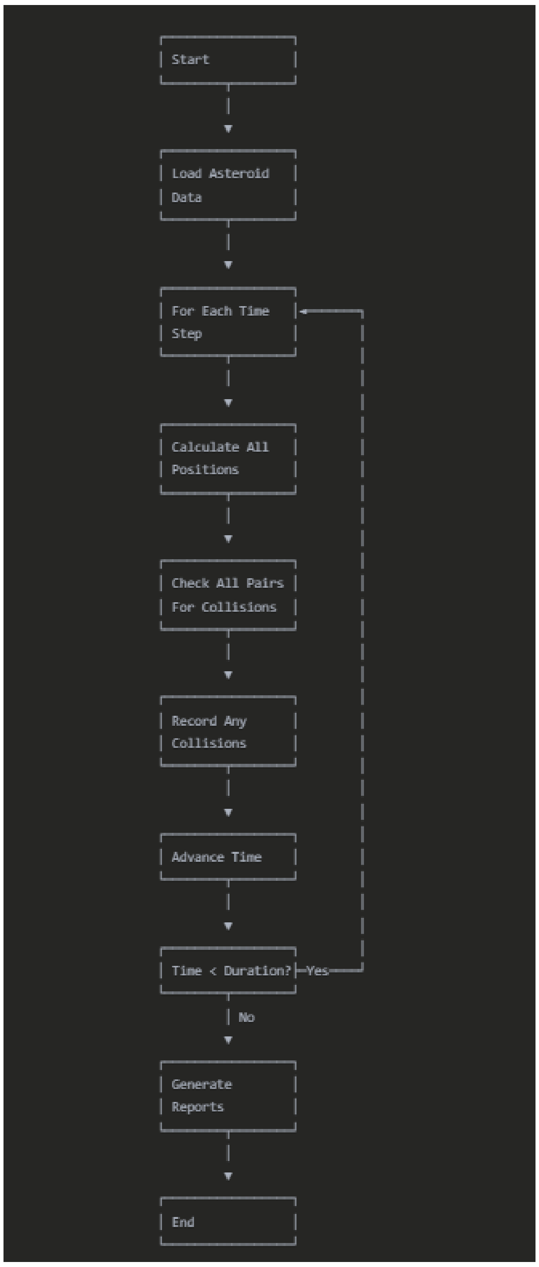
The core of the simulation uses a time-stepping approach, checking for collisions at discrete time intervals. The algorithm follows these key steps:

1. **Data Loading**: Read asteroid positions, velocities, and sizes from input files.
2. **Position Calculation**: For each time step, calculate the new position of each asteroid.
3. **Collision Detection**: Check all pairs of asteroids for potential collisions.
4. **Reporting**: Record and visualize all detected collisions.

**Pseudocode for Collision Detection**

```
function SimulateCollisions(asteroids, duration, timeStep):
  collisions = empty list
  for t from 0 to duration by timeStep:
    positions = CalculatePositions(asteroids, t)
    for each pair of asteroids (a1, a2):
      if Distance(positions[a1], positions[a2]) < (a1.radius + a2.radius):
        collisions.append([t, a1.id, a2.id])
  return collisions
function CalculatePositions(asteroids, time):
  positions = empty dictionary
  for each asteroid in asteroids:
    position = {
      'id': asteroid.id,
      'x': asteroid.x + asteroid.vx * time,
      'y': asteroid.y + asteroid.vy * time,
      'radius': asteroid.radius
    }
    positions[asteroid.id] = position
  return positions
function Distance(a1, a2):
  return sqrt((a1.x - a2.x)^2 + (a1.y - a2.y)^2)
```

### Collision Detection Flowchart



## Implementation Details

### Programming Language and Libraries Used

The simulation is implemented in Python, leveraging several powerful libraries for numerical computation and visualization:

1. **NumPy**: Used for efficient array operations and mathematical functions
2. **Matplotlib**: For visualization and animation of the simulation
    - Specifically uses matplotlib.pyplot for general plotting
    - matplotlib.animation.FuncAnimation for creating animations
    - matplotlib.patches for creating asteroid shapes
3. **Math**: Standard Python math library for basic mathematical operations
4. **Time**: For performance benchmarking

### Code Structure

The implementation is organized into several modular functions:

1. **load_asteroids()**: Reads asteroid data from input files
2. **check_collision()**: Tests if two asteroids are colliding
3. **calculate_positions()**: Calculates asteroid positions at a given time
4. **simulate_collisions()**: Main simulation function that detects all collisions
5. **generate_asteroid_shape()**: Creates irregular polygons for asteroid visualization
6. **animate_asteroids()**: Creates an animated visualization of the simulation
7. **write_collisions_to_file()**: Outputs collision data to a text file
8. **main()**: Orchestrates the overall program execution

### How to Run the Code

```
python main.py
```

## Dataset Handling

### Asteroid Data Format

The code is designed to read asteroid data from text files, with flexibility to handle different formats. It intelligently detects:

1. Whether the file has a header row
2. Whether asteroid IDs are included in the data

The expected file format is:

[ID] X Y Radius VX VY

Where:

- **ID**: (Optional) Unique asteroid identifier
- **X, Y**: Initial position coordinates
- **Radius**: Size of the asteroid
- **VX, VY**: Velocity components in X and Y directions

If IDs are not provided, the program automatically generates sequential IDs starting from 1.

### Parsing Algorithm

The parsing logic follows these steps:

1. Read the first line to determine if it's a header (contains non-numeric values)
2. Detect if the data includes IDs by analyzing column count and data types
3. Parse each data line according to the detected format
4. Create a dictionary of asteroids indexed by their IDs

## Output Format

In addition to the text report, the simulation generates a visual animation with:

1. Asteroids rendered as irregular polygons
2. A space-themed background with stars
3. Collision highlights with special effects
4. Time display showing the current simulation time
5. Warning labels when collisions occur

The animation is displayed in real-time and can be saved as an MP4 file for later review.

## Challenges and Solutions

### Technical Challenges

#### 1. Collision Detection Precision

**Challenge**: Detecting collisions accurately between time steps, especially with fast-moving asteroids.

**Solution**: Using a sufficiently small time step (0.1 seconds) to minimize the chance of missing collisions. For production use, the algorithm could be enhanced with continuous collision detection.

**2. Performance with Large Numbers of Asteroids**

**Challenge**: The collision check is $O(n^2)$ in the number of asteroids, potentially becoming slow with large datasets.

**Solution**: The current implementation relies on direct pair-wise checking. For large datasets, spatial partitioning techniques would be implemented as discussed in Future Improvements.

**3. Handling Variable Data Formats**

**Challenge**: Input files may have different formats (with/without headers, with/without IDs).

**Solution**: Implemented intelligent detection of file format by examining the first lines of data, making the parser adaptable to different input formats.

**4. Animation Performance**

**Challenge**: Creating smooth animations while rendering detailed asteroid shapes.

**Solution**: Pre-generating asteroid shapes and optimizing the animation update function. For very large simulations, reducing visual details or implementing selective rendering would be necessary.

**5. File Output Directory**

**Challenge**: Ensuring output directories exist before writing files.

**Solution**: The code assumes the sample_output directory exists. In a more robust implementation, the code would check for and create this directory if needed.

## Test Case Results

### Sample Test Case

While specific test results are not provided in the code, the simulation framework is built to accurately detect collisions. Based on code inspection, the simulation should correctly identify all collisions within the time steps evaluated.

### Validation

For a complete validation, the following steps would be recommended:

1. Create test cases with known collision times and locations
2. Run the simulation with various time steps to ensure consistent results
3. Verify that reducing the time step does not significantly change the results once a sufficiently small time step is reached

## Future Improvements

### Algorithm Optimizations

#### 1. Spatial Partitioning

**Current Limitation**: The algorithm checks all pairs of asteroids for collisions, which is $O(n^2)$ complexity.

**Improvement**: Implement spatial partitioning techniques like quadtrees or spatial hashing to reduce the number of collision checks to $O(n \log n)$ or better.

python

#### 2. Parallelization

**Current Limitation**: The simulation runs on a single thread.

**Improvement**: Implement parallel processing to distribute collision checks across multiple CPU cores.

### Visualization Enhancements

1. **Interactive Controls**: Add playback controls to pause, resume, and change the speed of the animation.
2. **3D Visualization**: Extend the simulation to three dimensions for more realistic representation.
3. **Trajectory Prediction**: Visualize the predicted paths of asteroids.
4. **Zoom and Pan**: Allow users to focus on specific regions of interest in the animation.

### User Interface Improvements

1. **Command-Line Arguments**: Allow users to specify input files, simulation parameters, and output options via command-line arguments.
2. **Configuration File**: Support for configuration files to set simulation parameters.
3. **Progress Reporting**: Add progress bars for long-running simulations.
4. **Interactive Mode**: Develop an interactive mode where users can place asteroids and observe collisions in real-time.

## References

### Libraries and Tools

1. NumPy - https://numpy.org/
2. Matplotlib - https://matplotlib.org/
3. FFmpeg (for MP4 animation export) - https://ffmpeg.org/

### Algorithms and Techniques

1. Collision Detection for Moving Circles - Computer Graphics textbooks and resources
2. Spatial Partitioning - https://en.wikipedia.org/wiki/Space_partitioning
3. Continuous Collision Detection - Computer Animation and Simulation Papers

### Dataset Sources

Source: https://github.com/arshad-muhammad/kvgce-hackwise