1 Introduction

This module models the physical environment of Titan, Saturn's largest moon, to support simulation of spacecraft descent and landing. The simulation includes a grid-based terrain model and atmospheric wind field. These are used to compute aerodynamic drag and altitude, essential for realistic landing dynamics and integration with physical simulation and optimization.

2 System Overview

The Titan environment is composed of several interconnected components:

- `CoordinateKey` – uniquely identifies grid cells by row and column.
- `PlanetSurfaceGrid` – converts global positions to grid coordinates.
- `PlanetHeightGrid` – stores terrain elevation per cell.
- `PlanetWindGrid` – stores wind vector per cell.
- `TitanEnvironment` – aggregates and exposes terrain/wind data.

Each module is designed to be independent, testable, and extensible. The grid resolution and domain are configurable via a fixed cell size.

2.1 CoordinateKey

The `CoordinateKey` class is a simple utility that defines a unique key for any cell in the grid using a pair of integers: `row` and `col`. It implements `equals()`, `hashCode()`, and `toString()` to enable efficient use in hash maps and debugging. This allows us to reference and cache values like terrain height or wind vector for any position in the simulation.

2.2 PlanetSurfaceGrid

This class is responsible for transforming 3D global positions into discrete grid keys. It uses the `x` and `z` components of a position vector (ignoring `y`, which represents altitude) and maps them to `CoordinateKey` objects by subtracting the planet center position and dividing by the `cellSize`. This ensures consistent spatial partitioning regardless of Titan's actual coordinates.

```
CoordinateKey toCoordinateKey(Vector3D pos) {
    double dx = pos.getX() - planetCenter.getX();
    double dz = pos.getZ() - planetCenter.getZ();
    int row = (int)Math.floor(dx / cellSize);
    int col = (int)Math.floor(dz / cellSize);
    return new CoordinateKey(row, col);
}
```

2.3 PlanetHeightGrid

This class maps each `CoordinateKey` to a terrain elevation value. We implemented two types of terrain generation:

- **Flat terrain** using `generateFlatTerrain(double baseHeight)`. This creates a uniform terrain height equal to the Y-position of Titan's center (e.g., `2.253e7` m).
- **Perlin terrain** using `generatePerlinTerrain(scale, amplitude, seed)`, which creates smooth hills and valleys based on a Perlin noise function.

Altitude queries are performed by calling `getAltitude(Vector3D position)`, which uses `PlanetSurfaceGrid` to determine the key and return the corresponding height.

2.4 PlanetWindGrid

Similar in structure to the height grid, this class maps `CoordinateKey` values to wind vectors. We implemented two generation modes:

- **Constant wind**: `generateConstantWind(Vector3D windVector)` assigns the same wind vector to every cell.
- **Perlin wind**: `generatePerlinWind(scale, maxWindSpeed, seed)` uses two Perlin noise maps (for X and Z directions) to assign natural-looking wind vectors across the surface.

Wind vectors are fetched using `getWind(Vector3D position)`.

2.5 TitanEnvironment

This class acts as a facade, aggregating access to both height and wind grids. It provides simple APIs:

- `getAltitude(Vector3D pos)`
- `getWind(Vector3D pos)`

It is the main point of interaction for higher-level components like the physics engine.

3 Atmospheric Drag Force

To model atmospheric effects on a spacecraft, we implemented the `AtmosphericForce` class. It computes the aerodynamic drag force exerted by Titan's atmosphere on a moving ship.

3.1 AtmosphericForce

The drag force is calculated using the equation:

$$\vec{F}_{drag} = -C_d \cdot v^2 \cdot \hat{v}_{rel}$$

where:

- C_d is the drag coefficient,
- v is the speed of the ship relative to wind,
- $\hat{v}_{rel}$ is the normalized relative velocity vector.

The class supports a maximum atmospheric altitude beyond which no drag is applied. The altitude is computed as:

```
double surface = environment.getAltitude(position);
double altitude = position.getY() - surface;
```

If `altitude > maxAtmosphereAltitude`, drag is zero. Otherwise, the drag is calculated using wind vectors from `TitanEnvironment`.

This ensures physically accurate behavior: no drag in space, increasing drag closer to the surface, and directionally correct forces.

4 Terrain and Wind Generation

We created utility methods for generating realistic planetary environments. The user can switch between flat and Perlin-based terrain, and between static or Perlin-based wind.

4.1 generateFlatTerrain(baseHeight)

Assigns a fixed height to all cells, typically Titan's surface Y-coordinate. Used for simple test cases.

4.2 generatePerlinTerrain(scale, amplitude, seed)

Uses a Perlin noise function to generate smooth terrain variation. Scale controls spatial frequency; amplitude controls height variation.

4.3 generateConstantWind(Vector3D)

Assigns the same wind vector across the surface — useful for debugging and baseline scenarios.

4.4 generatePerlinWind(scale, maxSpeed, seed)

Creates a smooth vector field using Perlin noise in X and Z directions. This allows spatially varying winds that simulate real atmospheric conditions on Titan.

5 Grid Projection vs. Spherical Representation

The current implementation models Titan's environment using a 2D grid projected onto the XZ-plane. This grid is centered on Titan's position and slices through the planet horizontally, much like a flat map laid across a sphere. Each cell corresponds to a square area (e.g., 10 km $\times$10 km), and grid keys are calculated based on the X and Z offsets from the planet's center.

This approach ignores the planet's curvature and treats the surface as a flat plane. While this is a simplification, it is highly effective for our primary use case: local descent and landing. Within a limited area (tens or hundreds of kilometers), the surface of Titan can be treated as flat without introducing significant error. This is similar to how airplanes navigate using flat maps over small regions of Earth.

A truly spherical model, using latitude-longitude or geodesic tiling, would be needed for global simulations (e.g., orbits or planetary coverage). Such a model would also increase implementation complexity and require spherical interpolation and conversion of coordinates.

Feature	**Flat Grid (current)**	**Spherical Model (ideal)**
Surface shape	Flat XZ plane	Curved sphere
Accuracy for descent	High (locally)	High (globally)
Coordinate system	Cartesian (x, z)	Spherical (lat/lon)
Code complexity	Low	High
Realism	Simplified	Accurate
Tested implementation	Yes	No

In summary, the current flat grid system is appropriate and efficient for localized simulations such as descent and landing. It simplifies both code and reasoning, while still producing realistic forces and terrain variation in the relevant region. A spherical model may be explored in the future to extend coverage to orbital and planetary-wide analyses.

6 Planned Integration

The Titan environment module is designed to integrate into the broader mission simulation pipeline. The next steps in the project involve extending this environment into the full physics simulation, flight control logic, and optimization framework. The integration plan is as follows:

1. **Thrust Modeling:** A propulsion model is being developed by other team members. Once completed, the spacecraft class will provide a time-dependent thrust vector, possibly based on dynamic control input or genetic parameters.

2. **Full Force Aggregation:** The PhysicsEngine will combine gravitational, thrust, and atmospheric drag forces using the following formula:

$$\vec{F}_{total} = \vec{F}_{gravity} + \vec{F}_{thrust} + \vec{F}_{drag}$$

   The drag term already comes from the AtmosphericForce module, which uses this TitanEnvironment system.

3. **Trajectory Evaluation:** Using the complete force model, the engine will simulate trajectories of the spacecraft and compute final landing metrics such as altitude, velocity, and remaining fuel.

4. **Landing Condition Verification:** The LandingCondition module will be used to determine whether a landing attempt is successful based on safety criteria such as speed and angle.

5. **Genetic Algorithm Integration:** The final component will involve connecting the simulation results to a genetic optimizer. Each individual trajectory will be scored using a landing fitness function, enabling iterative search for optimal descent parameters.

The Titan environment developed in this phase provides the foundation for accurate descent simulation and will be crucial in evaluating the viability of landing attempts in various atmospheric and terrain conditions.