API Documentation

Complete reference for the 3I/ATLAS Flight Tracker API, data formats, and component interfaces.

Table of Contents

Data Formats

TrajectoryData

Complete trajectory dataset loaded from JSON.

```
interface TrajectoryData {
  metadata: {
    generated: string;              // ISO timestamp of generation
    date_range: {
      start: string;                // Start date (YYYY-MM-DD)
      end: string;                  // End date (YYYY-MM-DD)
      current: string;              // Current date marker
    };
    step_size: string;              // Time step (e.g., "6h")
    units: {
      distance: string;             // "AU"
      velocity: string;             // "AU/day"
      time: string;                 // "ISO-8601"
    };
    source: string;                 // "NASA JPL Horizons System"
  };
  atlas: VectorData[];              // 3I/ATLAS trajectory points
  earth: VectorData[];              // Earth trajectory points
  mars: VectorData[];               // Mars trajectory points
  jupiter: VectorData[];            // Jupiter trajectory points
}
```

VectorData

Individual trajectory data point.

```typescript
interface VectorData {
  jd?: number;                      // Julian Date (optional)
  date: string;                     // ISO-8601 date string
  position: Vector3D;               // Position in AU
  velocity: Vector3D;               // Velocity in AU/day
  calculated?: boolean;             // True if calculated (not from API)
  distance_au?: number;             // Distance from Sun (optional)
  note?: string;                    // Additional notes (optional)
}
```

Vector3D

3D vector representation.

```typescript
interface Vector3D {
  x: number;                        // X coordinate (AU)
  y: number;                        // Y coordinate (AU)
  z: number;                        // Z coordinate (AU)
}
```

TimelineEvent

Event marker for key milestones.

```typescript
interface TimelineEvent {
  id: string;                       // Unique identifier
  name: string;                     // Display name
  date: string;                     // ISO-8601 date
  distance_au?: number;             // Distance at event (optional)
  max_velocity_kms?: number;        // Max velocity at event (optional)
  description: string;              // Short description
  type: 'milestone' | 'encounter';  // Event type
  educational_content?: string;     // Markdown content (optional)
}
```

Component APIs

Atlas3DTrackerEnhanced

Main visualization component.

```typescript
interface Atlas3DTrackerEnhancedProps {
  autoPlay?: boolean;               // Start playing automatically (default: true)
  initialSpeed?: number;            // Initial playback speed (default: 2)
  initialFollowMode?: boolean;      // Start in follow camera mode (default: true)
}

// Usage
<Atlas3DTrackerEnhanced
  autoPlay={true}
  initialSpeed={2}
  initialFollowMode={true}
/>
```

State Management:
- `trajectoryData` : Loaded trajectory data
- `currentIndex` : Current frame index
- `isPlaying` : Playback state
- `speed` : Current playback speed multiplier
- `followMode` : Camera follow mode enabled

FollowCamera

Camera that follows the comet.

```
interface FollowCameraProps {
  target: THREE.Vector3;            // Position to follow
  enabled: boolean;                 // Enable follow mode
  offset?: THREE.Vector3;           // Offset from target (default: [5,3,5])
  smoothness?: number;              // Interpolation factor (default: 0.05)
}

// Usage
<FollowCamera
  target={cometPosition}
  enabled={true}
  offset={new THREE.Vector3(5, 3, 5)}
  smoothness={0.05}
/>
```

CinematicCamera

Handles cinematic transitions for events.

```
interface CinematicCameraProps {
  active: boolean;                  // Trigger cinematic mode
  eventType: 'mars_flyby' | 'perihelion' | 'jupiter_approach' | null;
  target: THREE.Vector3;            // Camera target position
  onComplete?: () => void;          // Callback when animation completes
}

// Usage
<CinematicCamera
  active={true}
  eventType="perihelion"
  target={cometPosition}
  onComplete={() => setCinematicActive(false)}
/>
```

Comet3D

3D model of the comet.

```tsx
interface Comet3DProps {
  position: [number, number, number];  // XYZ position
  velocity: [number, number, number];  // XYZ velocity (for orientation)
  scale?: number;                      // Nucleus size (default: 0.05)
  tailLength?: number;                 // Tail length (default: 0.5)
}

// Usage
<Comet3D
  position={[1.5, 0, 0]}
  velocity={[0.01, 0.005, 0]}
  scale={0.05}
  tailLength={0.8}
/>
```

TrajectoryTrail

Renders the comet's path.

```tsx
interface TrajectoryTrailProps {
  trajectoryData: VectorData[];    // Full trajectory dataset
  currentIndex: number;            // Current playback position
  color?: string;                  // Trail color (default: "#00ff88")
  opacity?: number;                // Trail opacity (default: 0.8)
  lineWidth?: number;              // Line width (default: 2)
}

// Usage
<TrajectoryTrail
  trajectoryData={atlasData}
  currentIndex={500}
  color="#00ff88"
  opacity={0.8}
/>
```

TelemetryHUD

Real-time telemetry overlay.

```tsx
interface TelemetryHUDProps {
  currentFrame: VectorData | null;  // Current trajectory frame
  className?: string;               // Additional CSS classes
}

// Usage
<TelemetryHUD currentFrame={currentFrame} />
```

Displays:

- Current date
- Distance from Sun (AU and million km)
- Velocity (km/s and km/h)

PlaybackControls

User control interface.

```tsx
interface PlaybackControlsProps {
  isPlaying: boolean;               // Current play state
  speed: number;                    // Current speed multiplier
  currentIndex: number;             // Current frame index
  maxIndex: number;                 // Total frames
  followMode: boolean;              // Follow camera enabled
  onPlayPause: () => void;          // Play/pause callback
  onReset: () => void;              // Reset callback
  onSpeedChange: (speed: number) => void;  // Speed change callback
  onSeek: (index: number) => void;  // Seek callback
  onFollowModeToggle: () => void;   // Toggle follow mode callback
}

// Usage
<PlaybackControls
  isPlaying={isPlaying}
  speed={speed}
  currentIndex={currentIndex}
  maxIndex={trajectoryData.atlas.length - 1}
  followMode={followMode}
  onPlayPause={() => setIsPlaying(!isPlaying)}
  onReset={() => setCurrentIndex(0)}
  onSpeedChange={setSpeed}
  onSeek={setCurrentIndex}
  onFollowModeToggle={() => setFollowMode(!followMode)}
/>
```

TimelinePanel

Interactive event timeline.

```tsx
interface TimelinePanelProps {
  events: TimelineEvent[];          // Array of events
  onEventClick: (event: TimelineEvent) => void;  // Click callback
  className?: string;               // Additional CSS classes
}

// Usage
<TimelinePanel
  events={eventsData}
  onEventClick={(event) => {
    jumpToDate(event.date);
    showEducationalContent(event);
  }}
/>
```

Planet

Render a planet with orbit.

```tsx
interface PlanetProps {
  name: string;                    // Planet name
  trajectoryData: VectorData[];    // Planet trajectory
  currentIndex: number;            // Current position index
  radius: number;                  // Planet radius (AU)
  color: string;                   // Planet color
  showOrbit?: boolean;             // Show orbital path (default: true)
}

// Usage
<Planet
  name="Mars"
  trajectoryData={marsData}
  currentIndex={frameIndex}
  radius={0.04}
  color="#ff6666"
  showOrbit={true}
/>
```

Starfield

Animated background stars.

```tsx
interface StarfieldProps {
  count?: number;                  // Number of stars (default: 5000)
  radius?: number;                 // Sphere radius (default: 100)
  depth?: number;                  // Depth variation (default: 50)
}

// Usage
<Starfield count={3000} radius={80} depth={40} />
```

Hooks

useTrajectoryData

Custom hook for loading trajectory data.

```
function useTrajectoryData(url: string) {
  const [data, setData] = useState<TrajectoryData | null>(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<Error | null>(null);

  useEffect(() => {
    fetch(url)
      .then(res => res.json())
      .then(setData)
      .catch(setError)
      .finally(() => setLoading(false));
  }, [url]);

  return { data, loading, error };
}

// Usage
const { data, loading, error } = useTrajectoryData('/data/trajectory_static.json');
```

useAnimationFrame

Hook for custom animation loops.

```
function useAnimationFrame(callback: (deltaTime: number) => void, deps: any[]) {
  const requestRef = useRef<number>();
  const previousTimeRef = useRef<number>();

  useEffect(() => {
    const animate = (time: number) => {
      if (previousTimeRef.current !== undefined) {
        const deltaTime = time - previousTimeRef.current;
        callback(deltaTime);
      }
      previousTimeRef.current = time;
      requestRef.current = requestAnimationFrame(animate);
    };

    requestRef.current = requestAnimationFrame(animate);
    return () => cancelAnimationFrame(requestRef.current!);
  }, deps);
}
```

Utility Functions

Coordinate Conversion

```typescript
// Convert Horizons coordinates to Three.js coordinates
function horizonsToThreeJS(horizonsPos: Vector3D): [number, number, number] {
  return [
    horizonsPos.x,
    horizonsPos.z,
    -horizonsPos.y
  ];
}

// Convert AU to kilometers
const AU_TO_KM = 149597870.7;
function auToKm(au: number): number {
  return au * AU_TO_KM;
}

// Convert AU/day to km/s
function auPerDayToKmPerSec(auPerDay: number): number {
  return (auPerDay * AU_TO_KM) / 86400;
}
```

Distance Calculations

```typescript
// Calculate distance from origin
function distanceFromOrigin(pos: Vector3D): number {
  return Math.sqrt(pos.x ** 2 + pos.y ** 2 + pos.z ** 2);
}

// Calculate velocity magnitude
function velocityMagnitude(vel: Vector3D): number {
  return Math.sqrt(vel.vx ** 2 + vel.vy ** 2 + vel.vz ** 2);
}
```

Date Utilities

```
// Parse ISO date string
function parseISODate(dateStr: string): Date {
  return new Date(dateStr);
}

// Format date for display
function formatDate(dateStr: string): string {
  return new Date(dateStr).toLocaleDateString('en-US', {
    year: 'numeric',
    month: 'short',
    day: 'numeric',
  });
}

// Find closest frame index for a given date
function findFrameIndexByDate(
  trajectory: VectorData[],
  targetDate: Date
): number {
  return trajectory.findIndex(frame => {
    const frameDate = new Date(frame.date);
    return frameDate >= targetDate;
  });
}
```

Backend API

Python Script API

generate_atlas_trajectory.py

Command Line Interface:

```
# Generate static data
python3 generate_atlas_trajectory.py

# Force regeneration
python3 generate_atlas_trajectory.py --force

# Poll for updates
python3 generate_atlas_trajectory.py --poll

# Generate only event markers
python3 generate_atlas_trajectory.py --events-only
```

Python API:

```python
from generate_atlas_trajectory import TrajectoryDataGenerator

# Initialize generator
generator = TrajectoryDataGenerator()

# Generate static data
trajectory_data = generator.generate_static_data(force_api=False)

# Generate event markers
generator.generate_event_markers()

# Poll for updates
generator.poll_for_updates()
```

HorizonsAPIClient:

```python
from generate_atlas_trajectory import HorizonsAPIClient

client = HorizonsAPIClient()

# Look up object
result = client.lookup_object("C/2025 N1")

# Fetch vectors
vectors = client.fetch_vectors(
    command="1004083",
    start_date="2025-07-01",
    stop_date="2025-10-31",
    step_size="6h",
    center="@sun"
)
```

OrbitalMechanicsCalculator:

```python
from generate_atlas_trajectory import OrbitalMechanicsCalculator

calc = OrbitalMechanicsCalculator()

# Calculate position for a date
position = calc.calculate_position("2025-10-29")

# Generate full trajectory
trajectory = calc.generate_fallback_trajectory(
    start_date="2025-07-01",
    end_date="2025-10-31",
    hours_step=6
)

# Generate planet orbit
earth_orbit = calc.generate_planet_orbit(
    planet_name="earth",
    start_date="2025-07-01",
    end_date="2025-10-31",
    hours_step=24
)
```

REST API (Future Enhancement)

Potential REST API endpoints for dynamic data:

GET /api/trajectory

Fetch trajectory data for date range.

```
GET /api/trajectory?start=2025-07-01&end=2025-10-31&object=atlas

Response:
{
  "atlas": VectorData[],
  "metadata": {...}
}
```

GET /api/events

Fetch timeline events.

```
GET /api/events

Response:
{
  "events": TimelineEvent[]
}
```

GET /api/current-position

Get real-time position.

```
GET /api/current-position?object=atlas&date=2025-10-20

Response:
{
  "date": "2025-10-20T12:00:00Z",
  "position": {"x": 1.5, "y": 0.2, "z": -0.1},
  "velocity": {"vx": 0.01, "vy": 0.005, "vz": -0.002},
  "distance_au": 1.52
}
```

Error Handling

Common Error Codes

```
enum ErrorCode {
  DATA_LOAD_FAILED = 'DATA_LOAD_FAILED',
  TRAJECTORY_PARSE_ERROR = 'TRAJECTORY_PARSE_ERROR',
  INVALID_DATE_RANGE = 'INVALID_DATE_RANGE',
  WEBGL_NOT_SUPPORTED = 'WEBGL_NOT_SUPPORTED',
}

interface AppError {
  code: ErrorCode;
  message: string;
  details?: any;
}
```

Error Handling Example

```
try {
  const response = await fetch('/data/trajectory_static.json');
  if (!response.ok) {
    throw new AppError({
      code: ErrorCode.DATA_LOAD_FAILED,
      message: 'Failed to load trajectory data',
      details: { status: response.status }
    });
  }
  const data = await response.json();
  return data;
} catch (error) {
  console.error('Error loading trajectory:', error);
  // Show user-friendly error message
}
```

Performance Monitoring

Metrics to Track

```typescript
interface PerformanceMetrics {
  fps: number;                    // Frames per second
  loadTime: number;               // Data load time (ms)
  renderTime: number;             // Render time per frame (ms)
  memoryUsage: number;            // Memory usage (MB)
  trajectoryPoints: number;       // Number of data points
}

// Usage
function usePerformanceMonitoring() {
  const [metrics, setMetrics] = useState<PerformanceMetrics>({
    fps: 60,
    loadTime: 0,
    renderTime: 0,
    memoryUsage: 0,
    trajectoryPoints: 0,
  });

  useFrame((state) => {
    setMetrics(prev => ({
      ...prev,
      fps: Math.round(1 / state.clock.getDelta()),
      renderTime: state.clock.getDelta() * 1000,
    }));
  });

  return metrics;
}
```

Version History

- **v1.0.0** (October 2025)
- Initial release
- Full 3D visualization
- NASA Horizons integration
- Cinematic camera transitions
- Educational content integration

Support

For API questions or issues:
1. Check this documentation
2. Review component source code
3. Open an issue on GitHub
4. Contact development team

Happy coding! 🚀