User's Manual
for
SRM Orientation & Vector Transformations

Version 2.0, 18 November 2009

Table of Contents

List of Figures

1 Introduction

This user's manual describes how to use the Spatial Reference Model (SRM) software implementation to transform entity state information between different spatial reference frames. It provides an integrated example framework that is used to illustrate how software developers can use the SRM implementation to perform these transformation operations.

The remainder of this user's manual is organized as follows:

- Section 2 summarizes the scope of the SRM software addressed in this manual.
- Section 3 provides an overview of the relevant parts of the SRM Application Program Interface (API).
- Section 4 describes the framework used for the examples in this user's manual.
- Section 5 discusses the transformation of position information.
- Section 6 discusses the transformation of orientation information.
- Section 7 discusses the transformation of vector quantities, including linear velocity, angular velocity, linear acceleration, and angular acceleration information.

For background and general information on the SRM, please review the SRM standard (ISO/IEC 18026:2006), available from the SEDRIS web site at http://standards.sedris.org. This user's manual assumes the reader is familiar with the fundamental concepts and terminology of the SRM. In addition, familiarity with, and review of, the documentation contained within the SRM implementation software development kit (SDK) is recommended. Fundamental background information on the concept of orientation, the various forms in which orientation information can be represented, and the relationships between orientation, direction, and vector quantities is provided in the document "Technical Concepts: Orientation, Rotation, Velocity, and Acceleration and the SRM", which can be found at http://www.sedris.org/srm_desc.htm#papers.

2 Scope

The SRM software implementation provides facilities to transform entity state information for both particles and rigid bodies from one spatial reference frame (SRF) to another. The SRM implementation performs static conversions, but does **not** perform the kinematics calculations required to simulate the movements of the entities over time.

The types of entity state information that may be transformed using the SRM software implementation include:

- Position – the location of a particle, or of the center of mass of a rigid body, with respect to a specific spatial reference frame,
- Orientation – in general, the relationship between the axes of two linear spatial reference frames; in particular, the relationship between the coordinate axes of a rigid body, and the coordinate axes of a local tangent frame of a given world spatial reference frame,
- Velocity – the instantaneous rate of change of displacement (i.e., change of position) of a particle or rigid body,
- Angular Velocity – the instantaneous rate of rotation of a point or rigid body about an axis,
- Acceleration – the instantaneous rate of change of velocity of a particle or rigid body, and
- Angular Acceleration – the instantaneous rate of change of angular velocity of a point or rigid body.

Orientation may be represented in several ways, including axis-angle pairs, Euler angles, Tait-Bryan angles, 3x3 rotation matrices, and quaternions. Velocity, angular velocity, acceleration, and angular acceleration may be represented as vectors.

3 API Overview

This section provides a brief overview of the elements of the SRM API that are referred to in this user's manual. More complete and detailed information is contained in the SRM API documentation. Readers who are already familiar with the SRM API may prefer to skip this section.

The examples in this user's manual use the C++ language binding of the SRM API. Note that, for brevity, a number of elements that would be required in compilable source code are omitted. For example, the SRM namespace is assumed. Also, exception handling is ignored, although production code should always use try/catch blocks.

This user's manual makes use of a hierarchical collection of object classes and their associated methods. These are briefly summarized below.

Figure 3-1 summarizes the classes and methods in the portion of the SRM API class hierarchy that is referred to in this user's manual. As shown in the figure, these classes are organized into two hierarchies. The spatial reference frame (SRF) class hierarchy is shown on the left side of the figure. The orientation class hierarchy is shown on the right side of the figure.

3.1 SRF Classes

The SRF class hierarchy includes classes that represent various types of spatial reference frames. Note that Figure 3-1 shows only the subset of the classes defined by the SRM API that are referenced within this user's manual. See the SRM API documentation for a more comprehensive description of the SRF class hierarchy.

3.1.1 BaseSRF

The class `BaseSRF` is the abstract base class for all SRF classes. It provides several common methods to return coded values identifying the SRF and its associated Object Reference Model (ORM), Reference Transformation (RT), and Coordinate System (CS).

3.1.2 BaseSRF_3D

The class `BaseSRF_3D` is the abstract base class for all 3D SRF classes. It is a subclass of `BaseSRF`. It defines the following methods that are referenced within this user's manual:

1) `createCoordinate3D`,
2) `getCoordinate3DValues`,
3) `freeCoordinate3D`,
4) `changeCoordinate3DSRF`,
5) `createDirection`,
6) `freeDirection`,
7) `createLococentricEuclidean3DSRF`.
8) `transformOrientation`,
9) `transformOrientationCommonOrigin`,
10) `transformVector`,
11) `transformVectorCommonOrigin`,

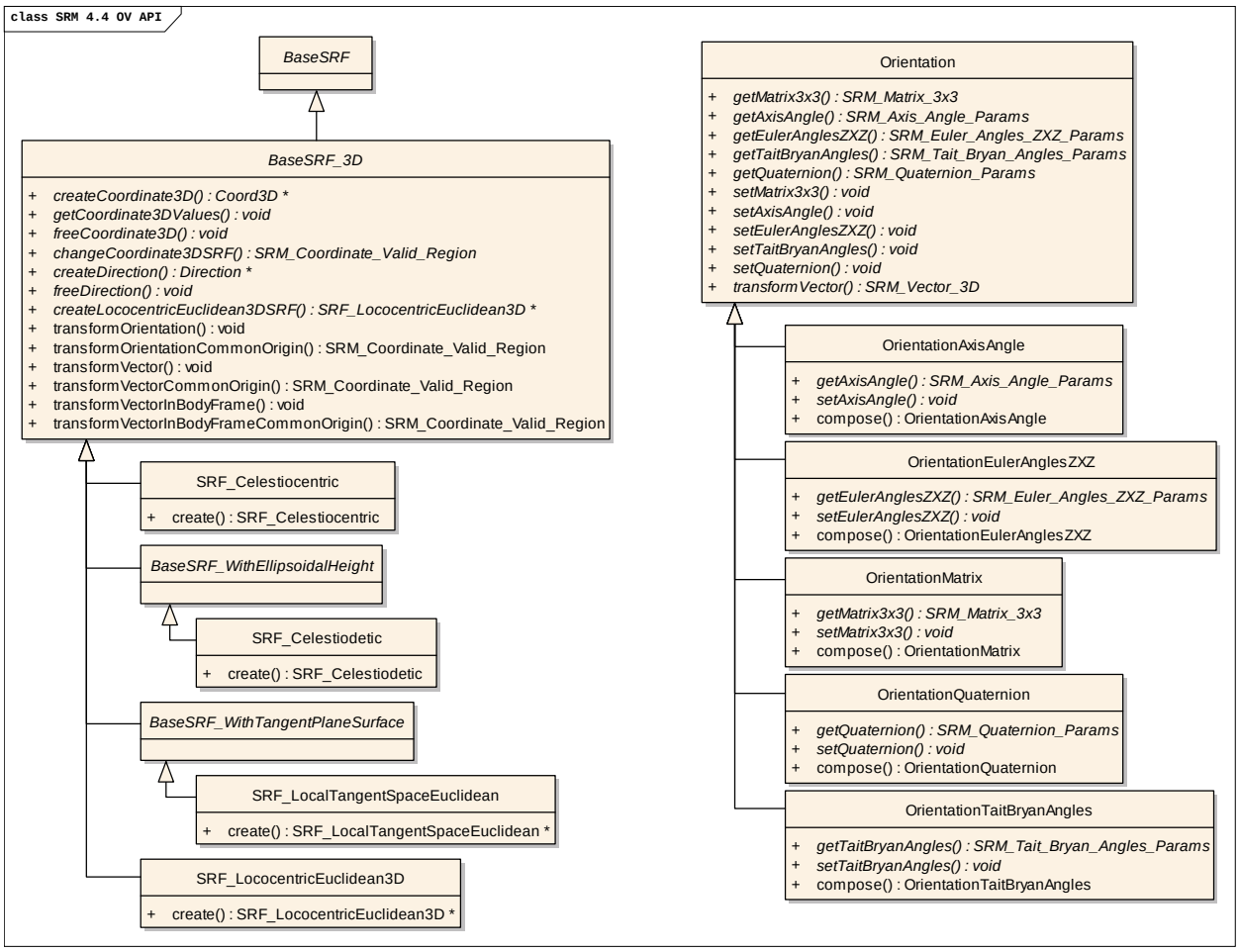

Figure 3-1. Subset of SRM Classes & Methods

12) `transformVectorInBodyFrame`, and
13) `transformVectorInBodyFrameCommonOrigin`.

It also defines a number of other methods. See the SRM API documentation for a complete description of these.

The methods `createCoordinate3D`, `getCoordinate3DValues`, `freeCoordinate3D`, and `changeCoordinate3DSRF` are used in dealing with position information. The method `createCoordinate3D` creates a `Coord3D` object from three coordinate component values that are specified as input parameters. The method `getCoordinate3DValues` outputs the three coordinate component values of a specified `Coord3D` object. The method `changeCoordinate3DSRF` transforms a specified `Coord3D` object from a specified source SRF to the target SRF. The method `freeCoordinate3D` releases a `Coord3D` object.

The methods `createDirection`, and `freeDirection` are used in dealing with direction information. The method `createDirection` creates a `Direction` object from a reference `Coord3D` object and a 3D vector that are specified as input parameters. The method `freeDirection` releases a `Direction` object.

The method `createLococentricEuclidean3DSRF` creates a `SRF_LococentricEuclidean3D` SRF relative to a given SRF, with its origin at a specified lococentre coordinate and its orientation determined by specified primary and secondary axis directions. It is used in the examples below to create platform body and component-based SRFs (see sections 4.2.2 and 4.2.3).

The methods `transformOrientation` and `transformOrientationCommonOrigin` are used in dealing with orientation information (see section 6). The method `transformOrientation` transforms a specified `Orientation` object with respect to a local tangent frame associated with a specified source reference coordinate in the source SRF, to the local tangent frame associated with the specified target reference coordinate in the target SRF. The method `transformOrientationCommonOrigin` is similar, but uses the same reference location for both the source and target SRFs, transforming the specified `Coord3D` object from the source SRF to the target SRF.

The methods `transformVector`, `transformVectorCommonOrigin`, `transformVectorInBodyFrame`, and `transformVectorInBodyFrameCommonOrigin` are used in dealing with vector quantities, such as velocity and acceleration information (see section 7). The method `transformVector` transforms a specified vector quantity with respect to a local tangent frame associated with a specified source reference coordinate in the source SRF, to the local tangent frame associated with a specified target reference coordinate in the target SRF. The method `transformVectorCommonOrigin` is similar, but uses the same reference location for both the source and target SRFs. The methods `transformVectorInBodyFrame` and `transformVectorInBodyFrameCommonOrigin` allow the source vector to be specified in terms of a body frame rather than a local tangent frame, by specifying the orientation of the body frame with respect to a local tangent frame.

3.1.3 SRF_Celestiocentric

The class `SRF_Celestiocentric` is derived from `BaseSRF3D` and defines SRFs that use a Euclidean 3D coordinate system in which the origin is located at the center of mass of a celestial body, the *xy*-plane is the plane of the equator, and the *xz*-plane contains the prime meridian. In this manual, it is used to define the Geocentric WGS 1984 SRF (see section 4.1.2).

3.1.4 BaseSRF_WithEllipsoidalHeight

The class `BaseSRF_WithEllipsoidalHeight` is an abstract class that is derived from `BaseSRF3D`. It is the parent class of `SRF_Celestiodetic` (see below), as well as other SRF classes that use coordinate systems based on the surface of an oblate ellipsoid.

3.1.5 SRF_Celestiodetic

The class `SRF_Celestiodetic` is derived from `BaseSRF3D` defines SRFs that use a geodetic 3D coordinate system, with coordinate-components longitude (λ) and latitude (φ), in radians, and ellipsoidal height (*h*), in meters. In this manual, it is used to define the Geodetic WGS 1984 SRF (see section 4.1.1).

3.1.6 BaseSRF_WithTangentPlaneSurface

The class `BaseSRF_WithTangentPlaneSurface` is an abstract class that is derived from `BaseSRF3D`. It is the parent class of `SRF_LocalTangentSpaceEuclidean` (see below), as well as other SRF classes that use coordinate systems based on planes tangent to the surface of an oblate ellipsoid.

3.1.7 SRF_LocalTangentSpaceEuclidean

The class `SRF_LocalTangentSpaceEuclidean` defines SRFs that use a Euclidean 3D coordinate system in which the *xy*-plane is tangent to the surface of the oblate ellipsoid that defines the Earth reference model. In this manual, it is used to describe local SRFs for test/training ranges. See section 4.2.1 for more details on this type of SRF.

3.1.8 SRF_LococentricEuclidean3D

The class `SRF_LococentricEuclidean3D` defines SRFs that use a Lococentric Euclidean 3D coordinate system. In this manual, it is used to define SRFs for individual entities such as tanks and aircraft, and individual entity components such as turrets (see sections 4.2.2 and 4.2.3).

3.2 Orientation Classes

The orientation class hierarchy consists of classes that represent the orientation of one SRF with respect to another. As shown in Figure 3-1, it consists of an abstract class with five concrete subclasses.

3.2.1 Orientation

The class `Orientation` represents the orientation of one SRF with respect to another. It provides methods that are common to all orientation representations. It provides the following methods that are referenced within this user's manual:

1) `getMatrix3x3`,
2) `getAxisAngle`,
3) `getEulerAnglesZXZ`,
4) `getTaitBryanAngles`,
5) `getQuaternion`,
6) `setMatrix3x3`,
7) `setAxisAngle`,
8) `setEulerAnglesZXZ`,
9) `setTaitBryanAngles`,
10) `setQuaternion`, and
11) `transformVector`.

The first five of these methods return the desired representation of the `Orientation` object (see 6.2). The next five methods allow the state of the `Orientation` object to be set using any of the five supported representations. The `Orientation` class also defines a number of other methods. See the SRM API documentation for a complete description of these.

The method `transformVector` transforms the representation of a three-dimensional vector from the source SRF of an `Orientation` object to the target SRF of that `Orientation` object.

The static method `compose`, which is implemented by each of the subclasses, composes two given `Orientation` objects and returns the resulting `Orientation` object. Thus, if S_1, S_2, and S_3 are three SRFs, $\texttt{Orientation}_{12}$ is the orientation of S_1 with respect to S_2, and $\texttt{Orientation}_{23}$ is the orientation of S_2 with respect to S_3, then the composition of these two orientations is $\texttt{Orientation}_{13}$, the orientation of S_1 with respect to S_3.

3.2.2 OrientationAxisAngle

The class `OrientationAxisAngle` is used to create and access orientation information using an axis-angle representation. See section 6.2.1 for more detail on this representation of orientations.

3.2.3 OrientationEulerAnglesZXZ

The class `OrientationEulerAnglesZXZ` is used to create and access orientation information using an Euler angle ZXZ representation. See section 6.2.2 for more detail on this representation of orientations.

3.2.4 OrientationTaitBryanAngles

The class `OrientationTaitBryanAngles` is used to create and access orientation information using a Tait-Bryan angle representation. See section 6.2.3 for more detail on this representation of orientations.

3.2.5 OrientationMatrix

The class `OrientationMatrix` is used to create and access orientation information using a 3x3 rotation matrix representation. See section 6.2.4 for more detail on this representation of orientations.

3.2.6 OrientationQuaternion

The class `OrientationQuaternion` is used to create and access orientation information using a quaternion representation. See section 6.2.5 for more detail on this representation of orientations.

4 Example Framework

This section presents the context for the examples that are presented throughout the subsequent sections of this document. These examples use several types of spatial reference frames, and several types of ground and airborne platforms, including some with articulated components, to illustrate the types of entity state transformations that may be performed using the SRM API. These examples do not reflect the full range of spatial reference frames supported by the SRM API, nor do they reflect the full range of entity types that can be represented.

The overall example framework is shown in Figure 4-1. Two test/training ranges are located adjacent to each other, each with its own spatial reference frame (SRF). The spatial extents of these two test/training range SRFs overlap somewhat. Note that, due to the curvature of the Earth's surface, the axes of these two SRFs are **not** parallel to each other. Ground forces are operating on both test/training ranges. One force (green) is operating primarily on Range 1, while another force (orange) is operating primarily on Range 2. Aircraft associated with both forces are operating overhead.

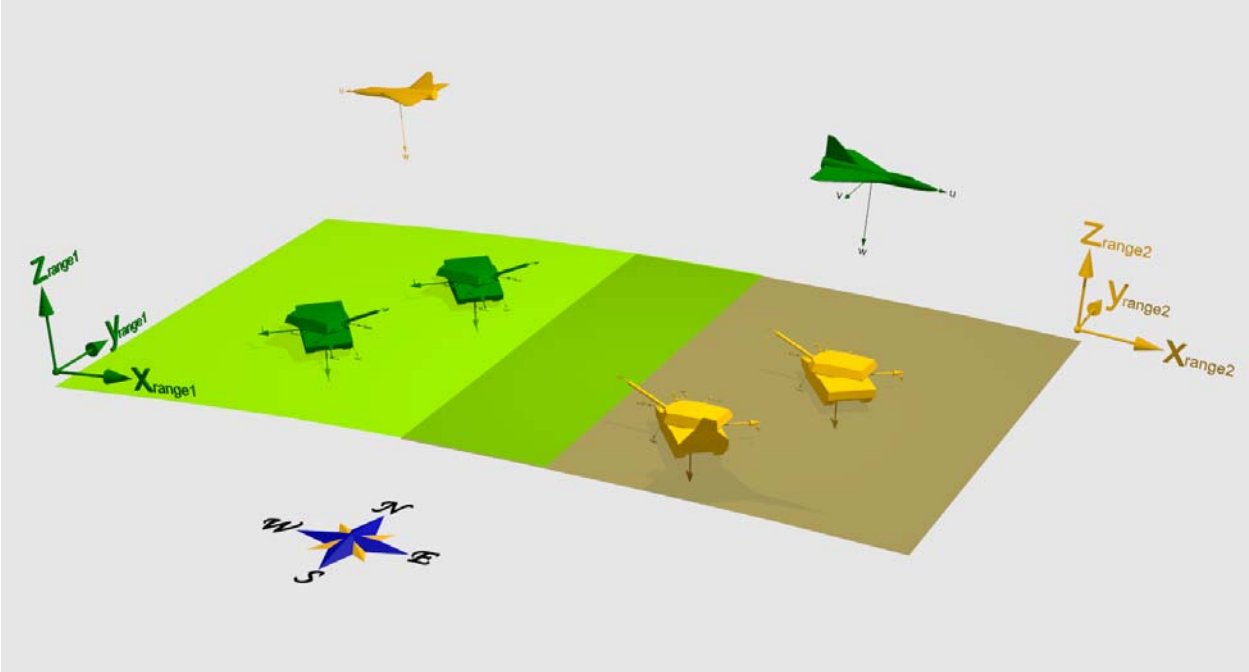

Figure 4-1. Example Framework

Each platform has its own local spatial reference frame. Associated with each aircraft is a local SRF, with axes defined in terms of the body of the aircraft. Similarly, each tank has a local hull SRF. Each tank turret and tank gun also has its own local SRF.

4.1 Global SRFs

To illustrate typical position, orientation, and vector quantity transformations, the examples in the subsequent sections use two global SRFs: the Geodetic WGS 1984 SRF; and the Geocentric WGS 1984 SRF. The Geodetic WGS 1984 SRF is used in the examples to specify the positions, orientations, and other state elements of various entities in global terms. The Geocentric WGS

1984 SRF is commonly used by applications to exchange entity state information using protocols such as the Distributed Interactive Simulation (DIS) standard[1], and so is also included in the examples.

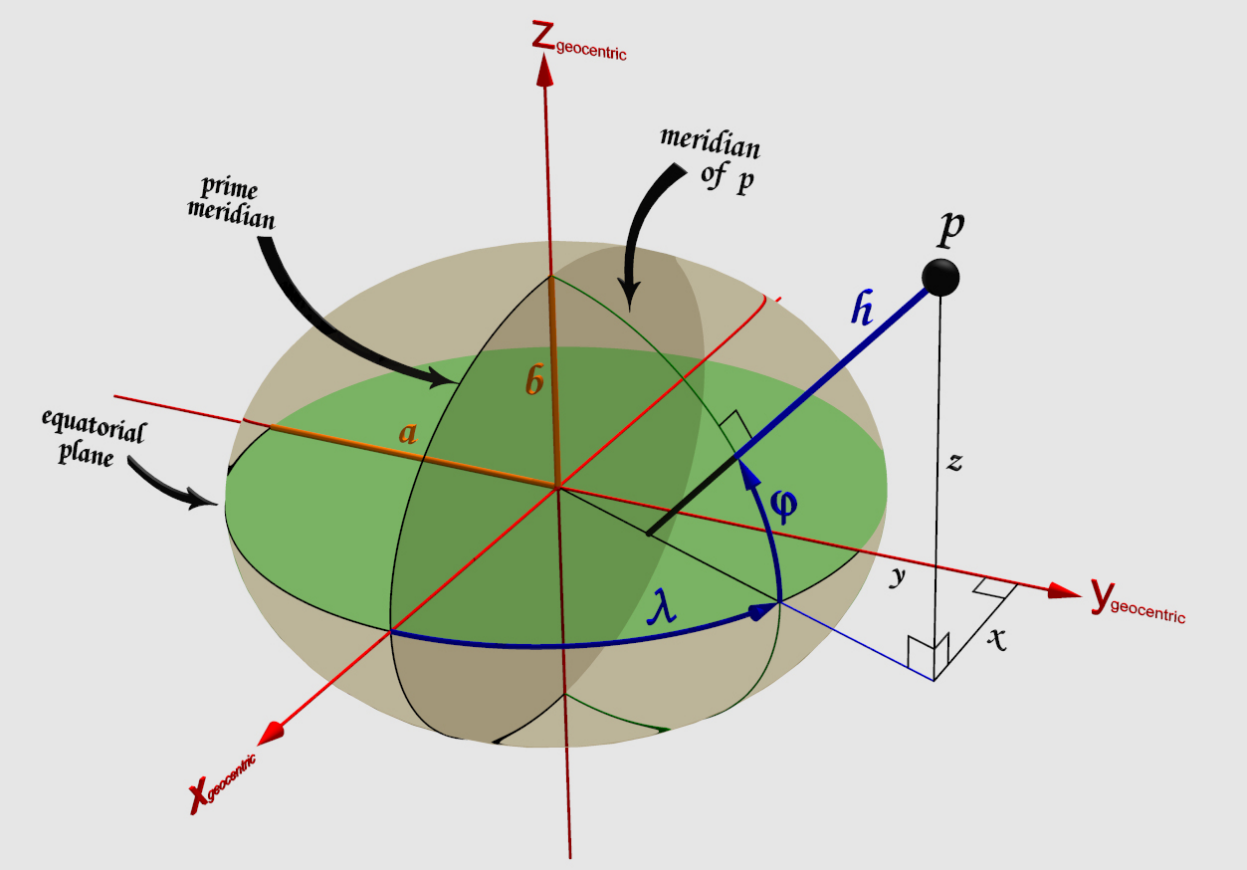

Figure 4-2. Global Geodetic and Geocentric SRFs

4.1.1 Geodetic WGS 1984 SRF

This is the standardized spatial reference frame identified by the SRM as `SRM_SRFCOD_GEODETIC_WGS_1984`. As shown in Figure 4-2, it uses a geodetic 3D coordinate system, with coordinate-components longitude (λ) and latitude (φ), in radians, and ellipsoidal height (h), in meters, with the World Geodetic System 1984 Earth reference model (`SRM_ORMCOD_WGS_1984`). The valid region for this SRF includes the surface of the oblate ellipsoid that represents the Earth and its vicinity.

Using the SRM API, this SRF may be instantiated as follows:

```
Geodetic_WGS84_SRF = SRF_Celestiodetic::create(
        SRM_ORMCOD_WGS_1984,  // Object Reference Model code
        SRM_RTCOD_WGS_1984_IDENTITY); // Reference Transformation code
```

[1] IEEE 1278.1-1995.

Note that this SRF is a curvilinear SRF, i.e., it is based on a curvilinear coordinate system that is referenced to the surface of the WGS 1984 ellipsoid. As a result, directions, orientations, and vector quantities that are specified with respect to this SRF are not independent of location. For example, a vector with component values (0, 0, 1), i.e., a vector pointing "up", denotes very different directions, depending on whether it is located at the North pole, at the intersection of the equator and the prime meridian, or at the South pole. Therefore, whenever a direction, orientation, or vector quantity is specified with respect to this SRF, or any other curvilinear SRF, a reference location must also be specified. This reference location serves as the origin of a Local Tangent Space Euclidean (LTSE) SRF, which provides the basis for the vector component values. The methods provided by the SRM API that are concerned with directions, orientations, and vector quantities include reference location parameters, which are required regardless of whether or not the SRF is curvilinear.

4.1.2 Geocentric WGS 1984 SRF

This is the standardized spatial reference frame identified by the SRM as `SRM_SRFCOD_GEOCENTRIC_WGS_1984`. As shown in Figure 4-2, it uses a Euclidean 3D coordinate system, with the axes *x*, *y*, and *z*, with the WGS 1984 Earth reference model. Its origin is located at the center of mass of the Earth. The *xy*-plane is the plane of the equator, while the *xz*-plane contains the prime meridian. The valid region of this SRF includes the surface of the oblate ellipsoid that represents the Earth and its vicinity.

Using the SRM API, this SRF may be instantiated as follows:

```
Geocentric_WGS84_SRF = SRF_Celestiocentric::create(
        SRM_ORMCOD_WGS_1984, // Object Reference Model code
        SRM_RTCOD_WGS_1984_IDENTITY);  // Reference Transformation code
```

This SRF is a linear SRF, i.e., it is based on a linear coordinate system. Directions, orientations, and vector quantities that are specified with respect to this SRF are independent of location. For example, a vector with component values (0, 0, 1), always denotes the same direction, i.e., the direction of the positive Z axis, regardless of its location. Therefore, whenever a reference location is required by the SRM API for a direction, orientation, or vector quantity that is specified with respect to this SRF, any convenient location may be chosen.

4.2 Local SRFs

Several types of local SRFs will also be addressed:

1) Test/training-range-based SRFs, which each are tied to a specified reference location within one of the test/training ranges.
2) Platform-based SRFs, which each are tied to the body of a specific moving platform, such as an aircraft or a tank.
3) Component-based SRFs, which each are tied to a particular component of a specific platform, such as the turret of a tank.

4.2.1 Test/Training-Range-Based SRFs

A local SRF is defined for each of the two test/training ranges in the examples. These two SRFs are illustrated in Figure 4-1. The Range 1 SRF has its origin located at the southwest corner of

the westernmost range. The Range 2 SRF has its origin located at the northeast corner of the easternmost range.

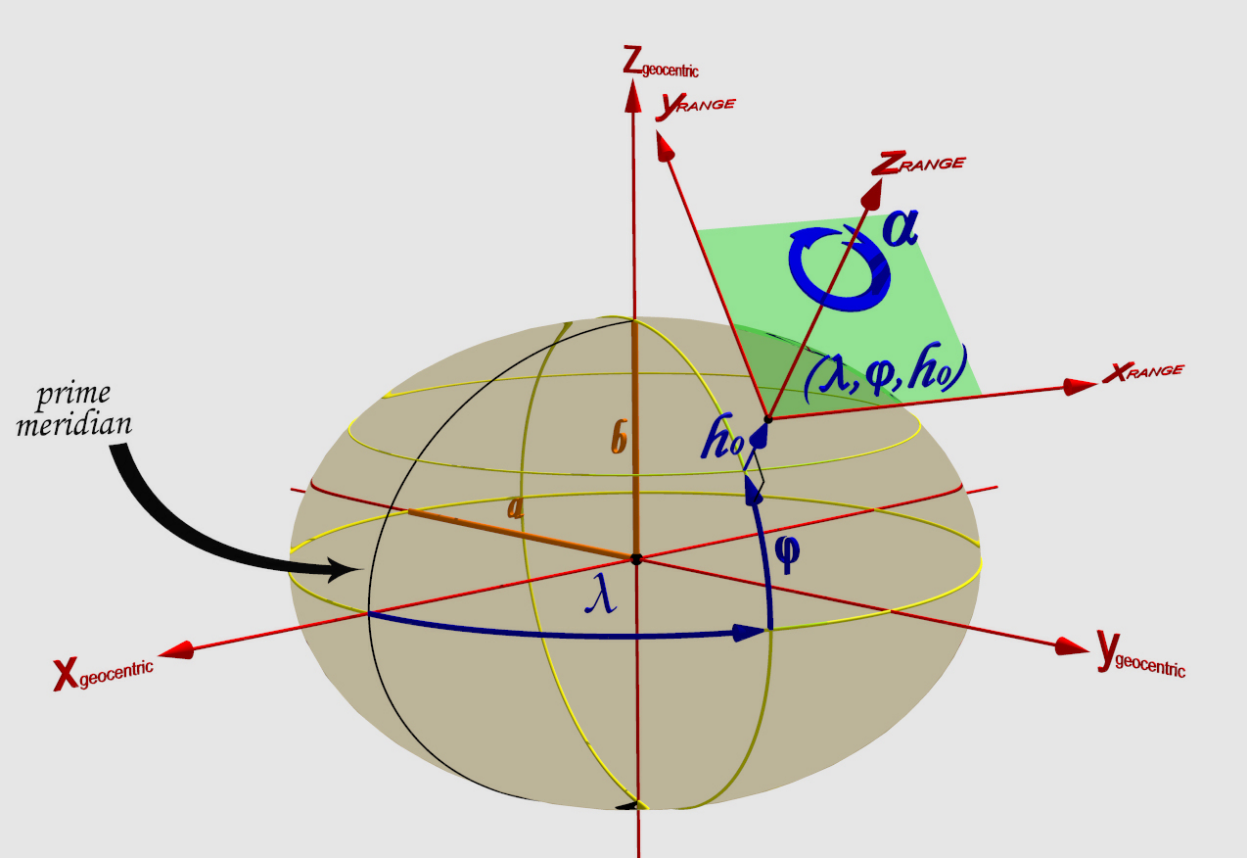

Figure 4-3. Local Tangent Space Euclidean SRF

These two SRFs are both instances of the Local Tangent Space Euclidean (LTSE) SRF template. These SRFs use a Euclidean 3D coordinate system, with the axes *x*, *y*, and *z*. As shown in Figure 4-3, the origin is tied to a reference location specified by a geodetic coordinate (λ, φ, h_0). The *xy* plane, shown in green, is parallel to the plane that is tangent to the ellipsoid surface of the WGS 1984 Earth reference model at the point (λ, φ). The angle α is the geodetic azimuth of the *y* axis. To avoid negative coordinate values, a false origin (x_F, y_F) may be specified.

Such local SRFs are linear SRFs, i.e., they are based on linear coordinate systems. Directions, orientations, and vector quantities that are specified with respect to such SRFs do not vary with location. Therefore, whenever a reference location is required by the SRM API for a direction, orientation, or vector quantity that is specified with respect to such an SRF, any convenient location, such as the origin of the SRF, may be chosen.

The ellipsoid heights of the origins of the two test/training range SRFs (see Figure 4-1) may differ from each other, since they are each located on the local terrain surface at their respective locations. In these SRFs, each positive *x* axis points to local east, while each positive *y* axis points to local north. The positive *z* axes point to local up. Note that, because of the curvature of the earth, the two *z* axes are not parallel to each other.

Suppose that the origin points of the two test/training ranges are separated by two degrees in longitude, and by one degree in latitude direction. The parameters for these test/training ranges could then be specified as:

Range 1 SRF:
- Origin longitude λ: -121°
- Origin latitude φ: 33°
- Origin ellipsoidal height h_0: 100 m
- Rotation angle α: 0°
- False origin x_F: 0 m
- False origin y_F: 0 m

Range 2 SRF:
- Origin longitude λ: -119°
- Origin latitude φ: 34°
- Origin ellipsoidal height h_0: 200 m
- Rotation angle α: 0°
- False origin x_F: 500,000 m
- False origin y_F: 500,000 m

The data type `SRM_LTSE_Parameters` contains the parameters that are used to specify an LTSE SRF. It includes parameters (longitude, latitude, and ellipsoidal height offset) that specify the origin location of the LTSE SRF. By default, the positive *x* axis points to local east, while the positive *y* axis points to local north. However, an azimuth parameter is included that can be used to explicitly rotate the *x*- and *y*-axes in the tangent plane. The height offset parameter is used to locate the origin above or below the surface of the ellipsoid. It also includes false origin offsets, so that negative coordinate values can be avoided.

```
typedef struct
{
    Long_Float geodetic_longitude; /* radians */
    Long_Float geodetic_latitude; /* radians */
    Long_Float azimuth; /* radians */
    Long_Float x_false_origin; /* meters */
    Long_Float y_false_origin; /* meters */
    Long_Float height_offset; /* meters */
} SRM_LTSE_Parameters;
```

Using the SRM API, these LTSE SRFs can be instantiated as follows:

```
SRM_LTSE_Parameters Range1_LTSE_Parameters;

Range1_LTSE_Parameters.geodetic_longitude = -121.0 * degreesToRadians;
Range1_LTSE_Parameters.geodetic_latitude = 33.0 * degreesToRadians;
Range1_LTSE_Parameters.azimuth = 0.0;
Range1_LTSE_Parameters.x_false_origin = 0.0;
Range1_LTSE_Parameters.y_false_origin = 0.0;
Range1_LTSE_Parameters.height_offset = 100.0;

//SRF_LocalTangentSpaceEuclidean* Range1_SRF;

Range1_SRF = SRF_LocalTangentSpaceEuclidean::create(
    SRM_ORMCOD_WGS_1984, // Object Reference Model code
    SRM_RTCOD_WGS_1984_IDENTITY, // Reference Transformation code
    Range1_LTSE_Parameters); // Local Tangent Space Euclidean params

SRM_LTSE_Parameters Range2_LTSE_Parameters;

Range2_LTSE_Parameters.geodetic_longitude = -119.0 * degreesToRadians;
```

```
Range2_LTSE_Parameters.geodetic_latitude = 34.0 * degreesToRadians;
Range2_LTSE_Parameters.azimuth = 0.0;
Range2_LTSE_Parameters.x_false_origin = 500000.0;
Range2_LTSE_Parameters.y_false_origin = 500000.0;
Range2_LTSE_Parameters.height_offset = 200.0;

//SRF_LocalTangentSpaceEuclidean* Range2_SRF;

Range2_SRF = SRF_LocalTangentSpaceEuclidean::create(
    SRM_ORMCOD_WGS_1984, // Object Reference Model code
    SRM_RTCOD_WGS_1984_IDENTITY, // Reference Transformation code
    Range2_LTSE_Parameters); // Local Tangent Space Euclidean params
```

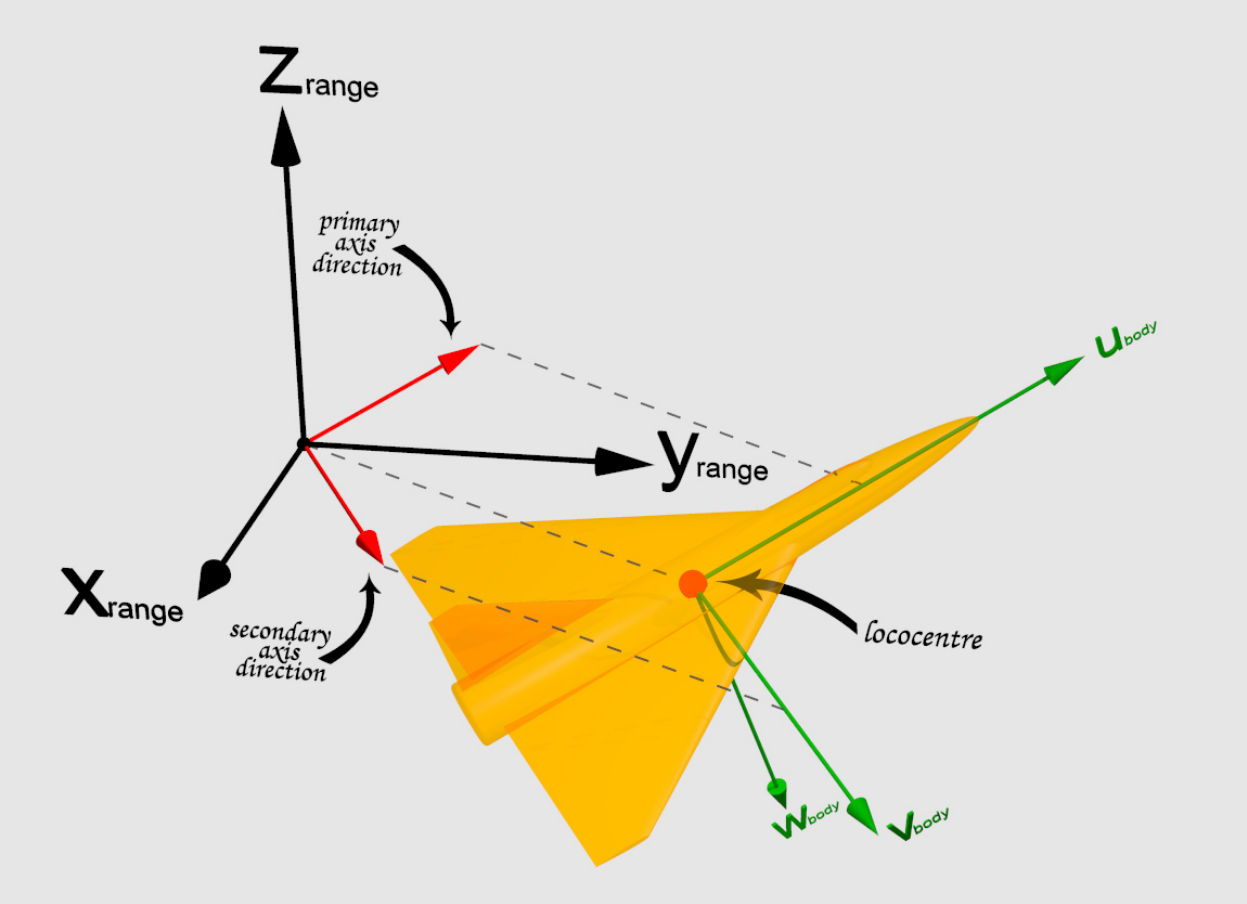

Figure 4-4. Aircraft Spatial Reference Frame

4.2.2 Platform-Based SRFs

For each moving platform, such as a tank or an aircraft, a platform-based SRF can be defined. These SRFs are instances of the `SRF_LococentricEuclidean3D` class. Such SRFs use a Lococentric Euclidean 3D coordinate system, with axis unit vectors ***u***, ***v***, and ***w***, and units in meters. As shown in Figures 4-4 and 4-5, the origin is specified by a lococentre, shown as a red sphere, which specifies the current position of the platform in the appropriate test/training range

SRF. The position of the platform is typically represented by its center of mass[2]. The orientation of the lococentric axes is determined by the current orientation of the platform's body or hull. This is specified by two direction vectors, also shown in red, with respect to the appropriate test/training range SRF. These primary and secondary axis direction vectors are parallel to the corresponding platform body axes. The axes of the lococentric platform-based SRFs are shown in green in Figures 4-4 and 4-5. The positive ***u*** axis points out the front of the platform. The positive ***v*** axis points out the right side of the platform. The positive ***w*** axis points down out of the bottom of the platform.[3]

Such platform-based SRFs are linear SRFs, i.e., they are based on linear coordinate systems. Directions, orientations, and vector quantities that are specified with respect to such SRFs do not vary with location. Therefore, whenever a reference location is required by the SRM API for a direction, orientation, or vector quantity that is specified with respect to such an SRF, any convenient location, such as the origin of the SRF, may be chosen.

The following example creates a platform-based SRF for a particular aircraft. Let ***c***, with components (c_x, c_y, c_z), be the coordinate of the aircraft body center of mass with respect to the `Range1_SRF` at a particular time. This location is the lococentre (or origin) for the aircraft body SRF (see Figure 4-4). Let the vectors ***p***, with components (p_x, p_y, p_z), and ***s***, with components (s_x, s_y, s_z), be unit vectors with respect to the `Range1_SRF`, which are parallel to the aircraft body primary and secondary axes, $\boldsymbol{u}_{body}$ and $\boldsymbol{v}_{body}$[4],. Using the SRM API, the aircraft body SRF can be instantiated as follows:

```
// Aircraft Body SRF
SRF_LococentricEuclidean3D* Body_SRF;
{
    SRM_Long_Float cx = 5000.0, cy = 10000.0, cz = 5000.0; // meters
    SRM_Long_Float px = -0.4330127, py = 0.75, pz = 0.5;
        // theta = 30 degrees, lambda = 120 degrees
    SRM_Long_Float sx = 0.8660254, sy = 0.5, sz = 0.0;

    // Lococentre
    Coord3D* lococentre = Range1_SRF->createCoordinate3D(cx, cy, cz);
    //Reference location for creating directions – using, for example,
    // the Range1_SRF origin
    Coord3D* ref_coord = Range1_SRF->createCoordinate3D(0.0, 0.0, 0.0);
    // Primary axis direction vector
    Direction* primary_axis_direction = Range1_SRF->createDirection(
        *ref_coord, px, py, pz);
    // Secondary axis direction vector
    Direction* secondary_axis_direction = Range1_SRF->createDirection(
        *ref_coord, sx, sy, sz);
```

[2] Note that, in general, the center of mass may not be static over time, as an aircraft expends fuel, jettisons munitions, etc. Also, note that the DIS protocol specifies the reference point to be the volumetric barycenter of the platform body, i.e., the geometric center of the volume that it occupies.

[3] This arrangement of the axes with respect to the platform body conforms to the DIS protocol.

[4] In the figure, these unit vectors have been lengthened to make them more visible, and to emphasize that they are parallel to the aircraft body axes.

```
        Body_SRF = Range1_SRF->createLococentricEuclidean3DSRF(
            *lococentre,
            *primary_axis_direction,
            *secondary_axis_direction);
    }
```

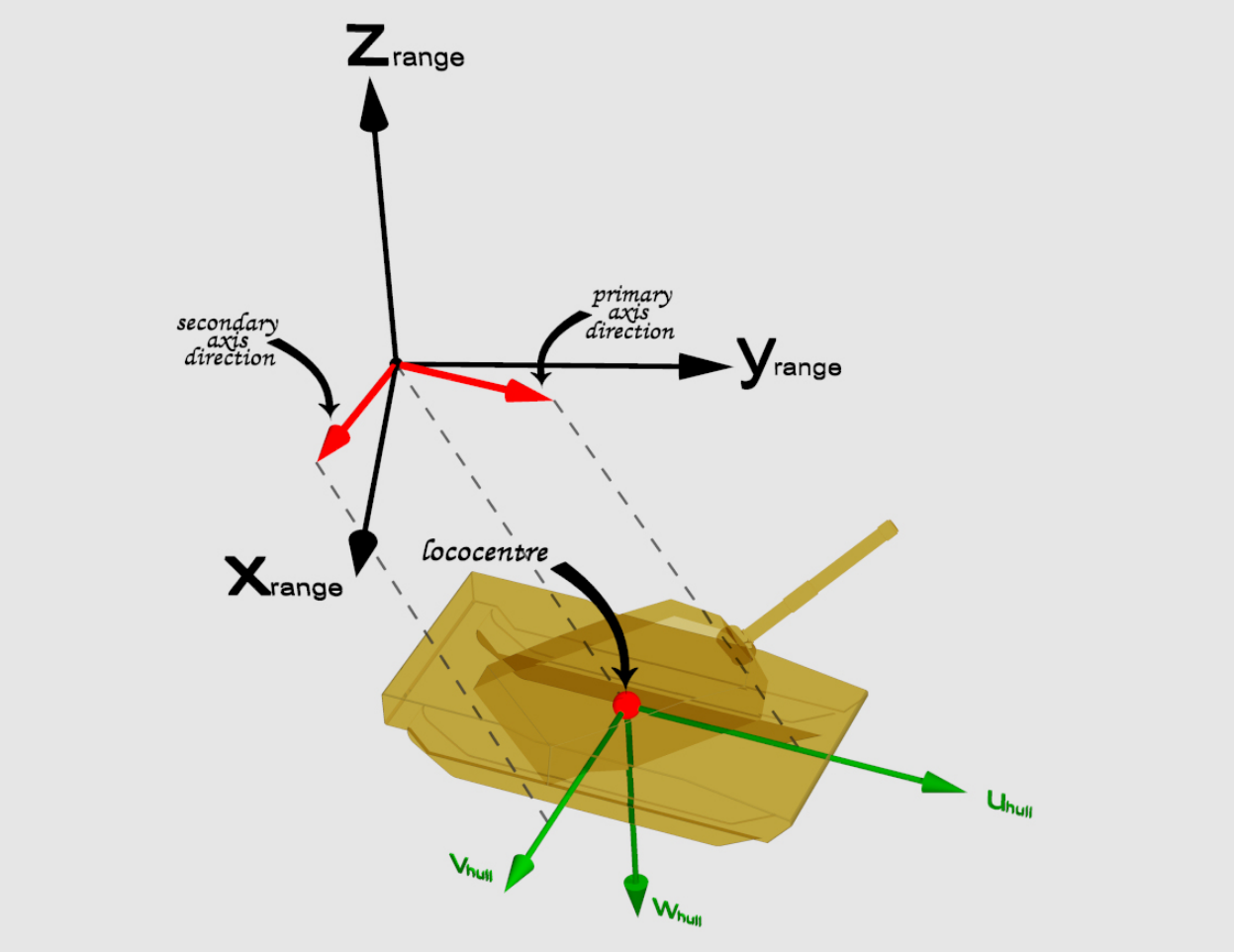

Figure 4-5. Tank Spatial Reference Frame

The next example creates a platform-based SRF for a particular tank. Let ***c***, with components (c_x, c_y, c_z), be the coordinate of the tank hull center of mass with respect to the `Range1_SRF` at a particular time. This location is the lococentre (or origin) for the tank hull SRF (see Figure 4-5). Let the vectors ***p***, with components (p_x, p_y, p_z), and ***s***, with components (s_x, s_y, s_z), be unit vectors with respect to the `Range1_SRF`, which are parallel to the tank hull primary and secondary axes, $\boldsymbol{u}_{hull}$ and $\boldsymbol{v}_{hull}$, respectively. Using the SRM API, the tank hull SRF can be instantiated as follows:

```
    // Tank Hull SRF
    SRF_LococentricEuclidean3D* Hull_SRF;
    {
        SRM_Long_Float cx = 2000.0, cy = 5000.0, cz = 500.0; // meters
        SRM_Long_Float px = 0.25, py = 0.9330127, pz = 0.25881905;
            // theta = 75 degrees, lambda = 15 degrees
        SRM_Long_Float sx = 0.96592583, sy = -0.25881905, sz = 0.0;
```

```
        // Lococentre
        Coord3D* lococentre = Range1_SRF->createCoordinate3D(cx, cy, cz);
        //Reference location for creating directions – using, for example,
        // the Range1_SRF origin
        Coord3D* ref_coord = Range1_SRF->createCoordinate3D(0.0, 0.0, 0.0);
        // Primary axis direction vector
        Direction* primary_axis_direction = Range1_SRF->createDirection(
            *ref_coord, px, py, pz);
        // Secondary axis direction vector
        Direction* secondary_axis_direction = Range1_SRF->createDirection(
            *ref_coord, sx, sy, sz);

        Hull_SRF = Range1_SRF->createLococentricEuclidean3DSRF(
            *lococentre,
            *primary_axis_direction,
            *secondary_axis_direction);

        Range1_SRF->freeDirection(secondary_axis_direction);
        Range1_SRF->freeDirection(primary_axis_direction);
        Range1_SRF->freeCoordinate3D(ref_coord);
        Range1_SRF->freeCoordinate3D(lococentre);
    }
```

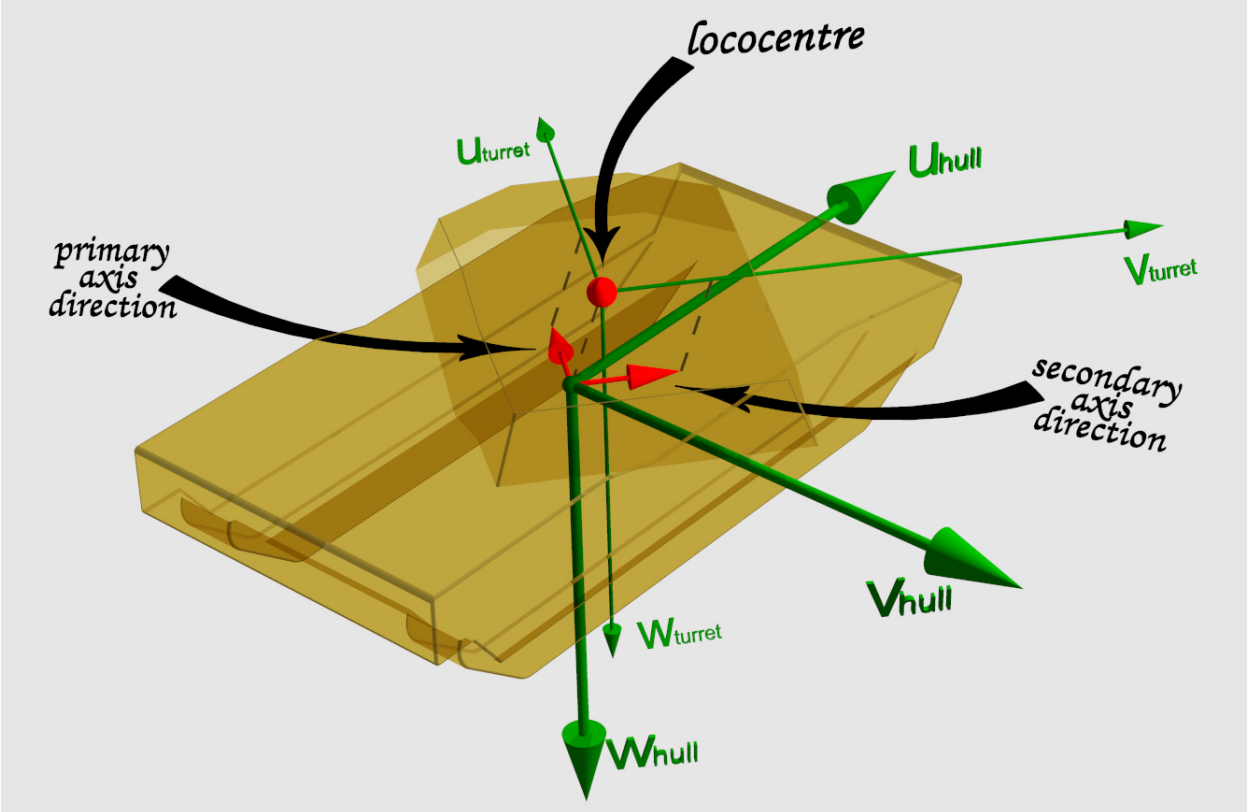

Figure 4-6. Tank Turret Spatial Reference Frame

4.2.3 Component-Based SRFs

Local SRFs can also be defined for articulated components of a platform. For example, for each tank, local SRFs can be defined for its turret, which rotates with respect to the hull, and for its gun, which can change its elevation angle with respect to the turret. Such SRFs are also instances of the Lococentric Euclidean 3D SRF template, which use a Lococentric Euclidean 3D coordinate system, with axis unit vectors ***u***, ***v***, and ***w***, and units in meters.

Such component-based SRFs are linear SRFs, i.e., they are based on linear coordinate systems. Directions, orientations, and vector quantities that are specified with respect to such SRFs do not vary with location. Therefore, whenever a reference location is required by the SRM API for a direction, orientation, or vector quantity that is specified with respect to such an SRF, any convenient location, such as the origin of the SRF, may be chosen.

The origin of the turret coordinate system is a point on the axis of rotation of the turret, specified in terms of the tank's local hull SRF. The orientation of the lococentric axes is determined by the current orientation of the turret, with respect to the hull. In Figure 4-6, the lococentre and the primary and secondary axis direction vectors that relate the turret coordinate system to the hull coordinate system are shown in red. The lococentric axes of the turret are shown in green with the subscript "turret". The positive ***u*** axis points out the front of the turret. The positive ***v*** axis points out the right side of the turret. The positive ***w*** axis points down out the bottom of the turret, parallel to the positive *w* axis of the hull. The turret coordinate system has only one degree of freedom, i.e., rotation about the turret's ***w*** axis.

The next example creates a local SRF for the turret of a particular tank. Let ***t***, with components (t_u, t_v, t_w), be the lococentre of the turret with respect to the `Hull_SRF` (see Figure 4-6). Let the vectors ***p***, with components (p_u, p_v, p_w), and ***s***, with components (s_u, s_v, s_w), be unit vectors with respect to the `Hull_SRF`, which are parallel to the turret primary and secondary axes, $\boldsymbol{u}_{turret}$ and $\boldsymbol{v}_{turret}$, respectively. Using the SRM API, the turret SRF can be instantiated as follows:

```
// Tank Turret SRF
SRF_LococentricEuclidean3D* Turret_SRF;
{
    SRM_Long_Float tu = 2.0, tv = 0.0, tw = -2.0;
    SRM_Long_Float pu = 0.70710678, pv = -0.70710678, pw = 0.0;
        // theta = 0 degrees, lambda = -45 degrees
    SRM_Long_Float su = 0.70710678, sv = 0.70710678, sw = 0.0;

    // Lococentre
    Coord3D* lococentre = Hull_SRF->createCoordinate3D(tu, tv, tw);
    //Reference location for creating directions – using, for example,
    // the Hull_SRF origin
    Coord3D* ref_coord = Hull_SRF->createCoordinate3D(0.0, 0.0, 0.0);
    // Primary axis direction vector
    Direction* primary_axis_direction = Hull_SRF->createDirection(
        *ref_coord, pu, pv, pw);
    // Secondary axis direction vector
    Direction* secondary_axis_direction = Hull_SRF->createDirection(
        *ref_coord, su, sv, sw);
    // Turret SRF
    Turret_SRF = Hull_SRF->createLococentricEuclidean3DSRF(
```

```
            *lococentre,
            *primary_axis_direction,
            *secondary_axis_direction);

        Hull_SRF->freeDirection(secondary_axis_direction);
        Hull_SRF->freeDirection(primary_axis_direction);
        Hull_SRF->freeCoordinate3D(ref_coord);
        Hull_SRF->freeCoordinate3D(lococentre);
    }
```

The origin of the gun coordinate system is the point along the long axis of the gun about which it rotates to change its elevation angle. The orientation of the axes of the gun coordinate system is determined by the current elevation angle of the gun, with respect to the turret. In Figure 4-7, the lococentre and the primary and secondary axis direction vectors that relate the gun coordinate system to the turret coordinate system are shown in red. The lococentric axes of the gun are shown in green with the subscript "gun". The positive ***u*** axis points out the barrel of the gun. The positive ***v*** axis points to the right, perpendicular to the gun barrel, and parallel to the positive ***v*** axis of the turret. The positive ***w*** axis points down, perpendicular to the gun barrel, as well as perpendicular to the ***v*** axis.

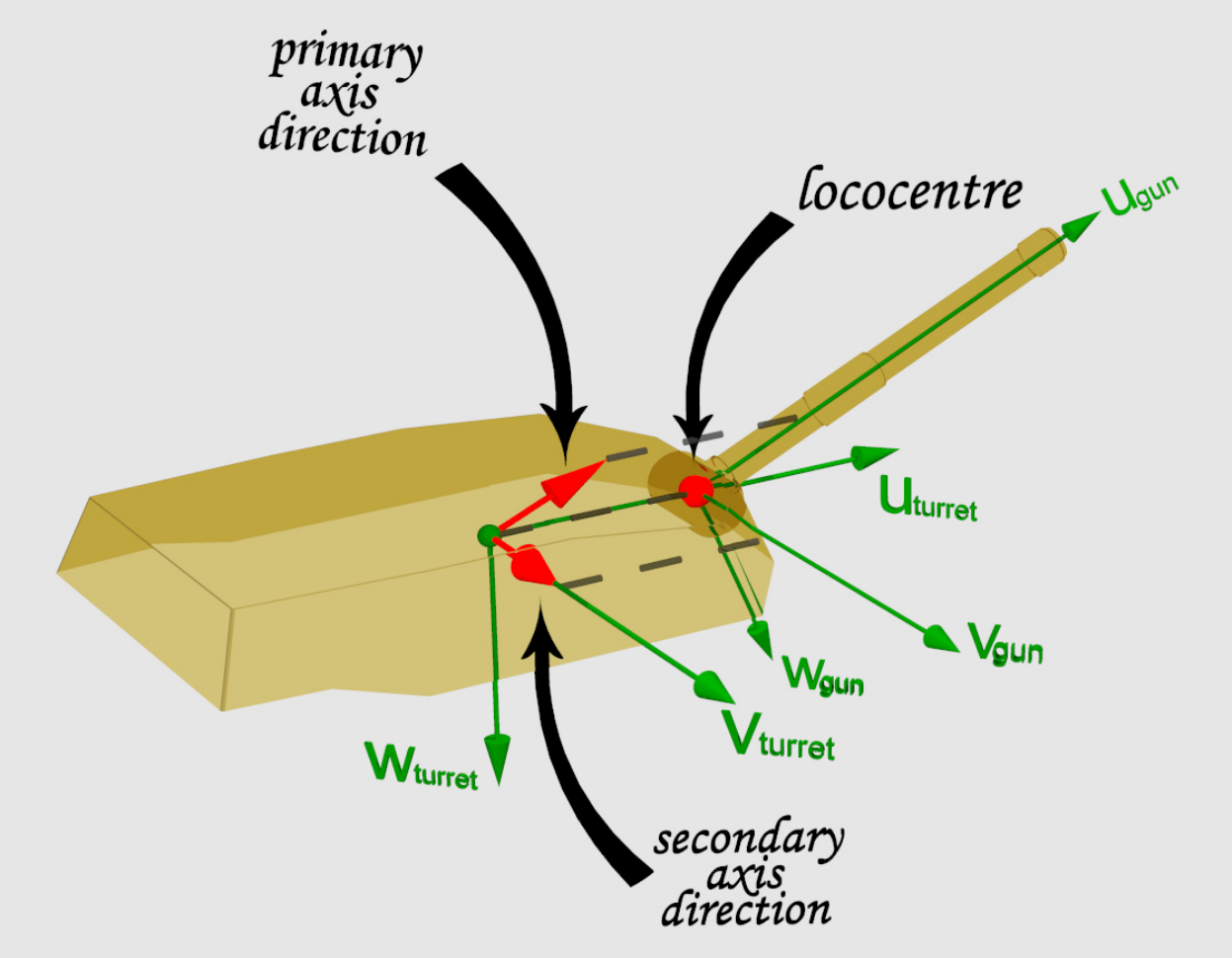

Figure 4-7. Tank Gun Spatial Reference Frame

The final example creates a local SRF for the gun of the tank. Let $\boldsymbol{g}$, with components (g_u, g_v, g_w), be the lococentre of the gun with respect to the `Turret_SRF` (see Figure 4-7). Let the vectors $\boldsymbol{p}$, with components (p_u, p_v, p_w), and $\boldsymbol{s}$, with components (s_u, s_v, s_w), be unit vectors with respect to the `Turret_SRF`, which are parallel to the gun primary and secondary axes, , $\boldsymbol{u}_{gun}$ and $\boldsymbol{v}_{gun}$, respectively. Using the SRM API, the gun SRF can be instantiated as follows:

```
// Tank Gun SRF
SRF_LococentricEuclidean3D* Gun_SRF;
{
    SRM_Long_Float gu = 3.0, gv = 0.0, gw = 0.0; // meters
    SRM_Long_Float pu = 0.96592583, pv = 0.0, pw = -0.25881905;
        // theta = -15 degrees, lambda = 0 degrees
    SRM_Long_Float su = 0.0, sv = 1.0, sw = 0.0;

    // Lococentre
    Coord3D* lococentre = Turret_SRF->createCoordinate3D(gu, gv, gw);
    //Reference location for creating directions – using, for example,
    // the Turret_SRF origin
    Coord3D* ref_coord = Turret_SRF->createCoordinate3D(0.0, 0.0, 0.0);
    // Primary axis direction vector
    Direction* primary_axis_direction = Turret_SRF->createDirection(
        *ref_coord, pu, pv, pw);
    // Secondary axis direction vector
    Direction* secondary_axis_direction = Turret_SRF->createDirection(
        *ref_coord, su, sv, sw);
    // Gun SRF
    Gun_SRF = Turret_SRF->createLococentricEuclidean3DSRF(
        *lococentre,
        *primary_axis_direction,
        *secondary_axis_direction);

    Turret_SRF->freeDirection(secondary_axis_direction);
    Turret_SRF->freeDirection(primary_axis_direction);
    Turret_SRF->freeCoordinate3D(ref_coord);
    Turret_SRF->freeCoordinate3D(lococentre);
}
```

5 Position

This section addresses the transformation of position information from one spatial reference frame to another.

5.1 Concept

Position refers to the location of a particle or rigid body with respect to a specific spatial reference frame, in either two or three dimensions. All of the spatial reference frames that will be used here are three-dimensional. As shown in Figure 5-1, position can be considered to be a vector from the origin of the SRF to the location of the particle or rigid body. It is usually symbolized as **p**. The coordinate axis names and the associated units vary, depending on the nature of the SRF.

The position of a rigid body is denoted by a representative point, which is usually, but not always, considered to be at the center of mass of that body.

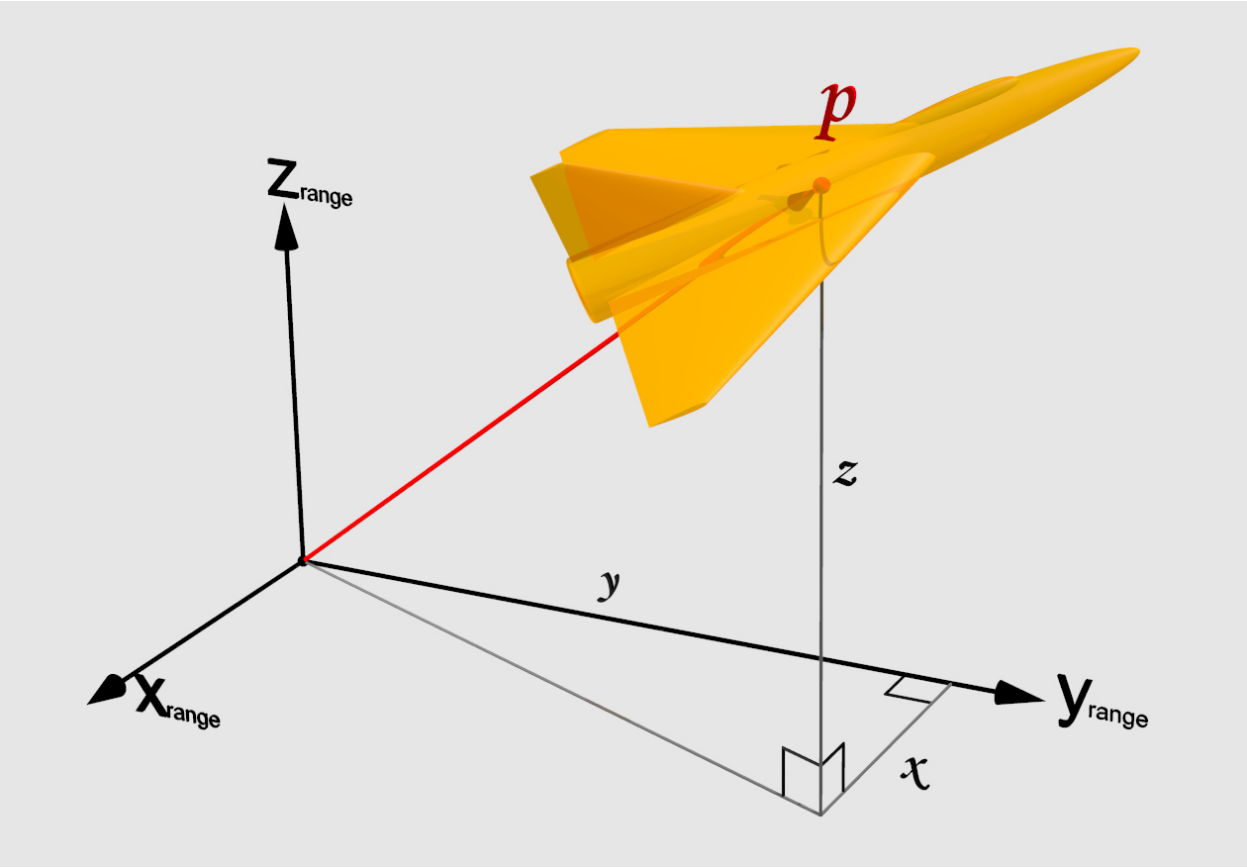

Figure 5-1. Position.

5.2 Transformation procedure

The general procedure for transforming a position from one SRF to another is:

1) Create the source SRF object, as shown in Section 4.
2) Create the target SRF object, as shown in Section 4.

3) Create a `Coord3D` object using the `createCoordinate3D` method of the source SRF object.
4) Obtain the equivalent `Coord3D` object for the target SRF by using the `changeCoordinate3D` method of the target SRF object.

The method `createCoordinate3D` creates a `Coord3D` object from three coordinate component values that are specified as input parameters. As used in this manual, it is invoked as follows:

```
Coord3D* coordinate = SRF->createCoordinate3D (
        first_coordinate_component, /* input, Long_Float */
        second_coordinate_component, /* input, Long_Float */
        third_coordinate_component); /* input, Long_Float */
```

The method `getCoordinate3DValues` outputs the three coordinate component values of a specified `Coord3D` object. As used in this manual, it is invoked as follows.

```
SRF->getCoordinate3DValues (coordinate, /* input, Coord3D */
        first_coordinate_component, /* output, Long_Float */
        second_coordinate_component, /* output, Long_Float */
        third_coordinate_component); /* output, Long_Float */
```

The method `changeCoordinate3DSRF` transforms a specified `Coord3D` object from a specified source SRF to the target SRF. As used in this manual, it is invoked as follows.

```
SRM_Coordinate_Valid_Region valid_region =
    target_SRF->changeCoordinate3DSRF (
        source_coordinate, /* input, Coord3D */
        target_coordinate); /* output, Coord3D */
```

The enumerated type `SRM_Coordinate_Valid_Region` describes the position of a coordinate with respect to the defined valid region of a specified SRF.

```
typedef enum
{
    SRM_COORDVALRGN_VALID,
    SRM_COORDVALRGN_EXTENDED_VALID,
    SRM_COORDVALRGN_DEFINED )
} SRM_Coordinate_Valid_Region;
```

Its possible values are:

`VALID` – the position is within the valid region of the specified SRF.

`EXTENDED_VALID` – the position is outside the valid region, but is within the extended valid region, of the specified SRF.

`DEFINED` – the position is not within either the valid region or the extended valid region of the specified SRF, but is in the domain of the coordinate system generating function.

5.3 Examples

The following examples show how to transform positions between the various SRFs defined in Section 4.

5.3.1 Transform Between Range SRFs

Example 1: Transform the position of a tank from the Range 1 SRF to the Range 2 SRF.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).

2) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
3) Create a `Coord3D` object using the `createCoordinate3D` method of the Range 1 SRF object.

```
SRM_Long_Float tank_range1_x = 2000.0, tank_range1_y = 5000.0,
    tank_range1_z = 500.0; // meters

Coord3D* range1_coordinate = Range1_SRF->createCoordinate3D(
    tank_range1_x,
    tank_range1_y,
    tank_range1_z);
```

4) Obtain the equivalent `Coord3D` object for the Range 2 SRF by using the `changeCoordinate3D` method of the Range 2 SRF object.

```
Coord3D* range2_coordinate = Range2_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

SRM_Coordinate_Valid_Region region = Range2_SRF->changeCoordinate3DSRF(
    *range1_coordinate,
    *range2_coordinate);

SRM_Long_Float tank_range2_x = 0.0, tank_range2_y = 0.0,
    tank_range2_z = 0.0;

Range2_SRF->getCoordinate3DValues(*range2_coordinate,
    tank_range2_x,
    tank_range2_y,
    tank_range2_z);
```

The output parameter `region` indicates the status of the resulting coordinate with respect to the valid region, and extended valid region, of the target SRF (see 5.2).

5.3.2 Transform From Range to Geocentric

Example 2: Transform the position of a tank from the Range 1 SRF to the Geocentric WGS 1984 SRF.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Geocentric WGS 1984 SRF (see Section 4.1.2).
3) Create a `Coord3D` object using the `createCoordinate3DSRF` method of the Range 1 SRF object.

```
SRM_Long_Float tank_range1_x = 2000.0, tank_range1_y = 5000.0,
    tank_range1_z = 500.0; // meters

Coord3D* range1_coordinate = Range1_SRF->createCoordinate3D(
    tank_range1_x,
    tank_range1_y,
    tank_range1_z);
```

4) Obtain the equivalent `Coord3D` object for the Geocentric WGS 1984 SRF by using the `changeCoordinate3D` method of the Geocentric WGS 1984 SRF object.

```
Coord3D* geocentric_coordinate =
    Geocentric_WGS84_SRF->createCoordinate3D(0.0, 0.0, 0.0);
```

```
SRM_Coordinate_Valid_Region region =
    Geocentric_WGS84_SRF->changeCoordinate3DSRF(
         *range1_coordinate,
         *geocentric_coordinate);

SRM_Long_Float tank_geocentric_x = 0.0, tank_geocentric_y = 0.0,
    tank_geocentric_z = 0.0;

Geocentric_WGS84_SRF->getCoordinate3DValues(*geocentric_coordinate,
    tank_geocentric_x,
    tank_geocentric_y,
    tank_geocentric_z);
```

5.3.3 Transform From Range to Geodetic

Example 3: Transform the position of a tank from the Range 2 SRF to the Geodetic WGS 1984 SRF.

1) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
2) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
3) Create a `Coord3D` object using the `createCoordinate3DSRF` method of the Range 2 SRF object.

```
SRM_Long_Float tank_range2_x = -5000.0, tank_range2_y = -3000.0,
    tank_range2_z = 400.0; // meters

Coord3D* range2_coordinate = Range2_SRF->createCoordinate3D(
    tank_range2_x,
    tank_range2_y,
    tank_range2_z);
```

4) Obtain the equivalent `Coord3D` object for the Geodetic WGS 1984 SRF by using the `changeCoordinate3D` method of the Geodetic WGS 1984 SRF object.

```
Coord3D* geodetic_coordinate =
    Geodetic_WGS84_SRF->createCoordinate3D(0.0, 0.0, 0.0);

SRM_Coordinate_Valid_Region region =
    Geodetic_WGS84_SRF->changeCoordinate3DSRF(
         *range2_coordinate,
         *geodetic_coordinate);

SRM_Long_Float tank_geodetic_longitude = 0.0,
    tank_geodetic_latitude = 0.0, tank_ellipsoidal_height = 0.0;

Geodetic_WGS84_SRF->getCoordinate3DValues(*geodetic_coordinate,
    tank_geodetic_longitude,
    tank_geodetic_latitude,
    tank_ellipsoidal_height);
```

5.3.4 Transform From Geodetic to Geocentric

Example 4: Transform the position of an aircraft from the Geodetic WGS 1984 SRF to the Geocentric WGS 1984 SRF.

1) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
2) Create the SRF object for the Geocentric WGS 1984 SRF (see Section 4.1.2).
3) Create a `Coord3D` object using the `createCoordinate3DSRF` method of the Geodetic WGS 1984 SRF object.

```
SRM_Long_Float aircraft_longitude = -120.5 * degreesToRadians,
    aircraft_latitude = 33.5 * degreesToRadians,
    aircraft_ellipsoidal_height = 5000.0; // meters

Coord3D* geodetic_coordinate = Geodetic_WGS84_SRF->createCoordinate3D(
    aircraft_longitude,
    aircraft_latitude,
    aircraft_ellipsoidal_height);
```

4) Obtain the equivalent `Coord3D` object for the Geocentric WGS 1984 SRF by using the `changeCoordinate3D` method of the Geocentric WGS 1984 SRF object.

```
Coord3D* geocentric_coordinate =
    Geocentric_WGS84_SRF->createCoordinate3D(0.0, 0.0, 0.0);

SRM_Coordinate_Valid_Region region =
    Geocentric_WGS84_SRF->changeCoordinate3DSRF(
        *geodetic_coordinate,
        *geocentric_coordinate);

SRM_Long_Float aircraft_x = 0.0, aircraft_y = 0.0, aircraft_z = 0.0;

Geocentric_WGS84_SRF->getCoordinate3DValues(*geocentric_coordinate,
    aircraft_x,
    aircraft_y,
    aircraft_z);
```

5.3.5 Transform From Geodetic to Range

Example 5: Transform the position of an aircraft from the Geodetic WGS 1984 SRF to the Range 2 SRF.

1) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
2) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
3) Create a `Coord3D` object using the `createCoordinate3DSRF` method of the Geodetic WGS 1984 SRF object.

```
SRM_Long_Float aircraft_longitude = -120.5 * degreesToRadians,
    aircraft_latitude = 33.5 * degreesToRadians,
    aircraft_ellipsoidal_height = 5000.0; // meters

Coord3D* geodetic_coordinate = Geodetic_WGS84_SRF->createCoordinate3D(
    aircraft_longitude,
    aircraft_latitude,
    aircraft_ellipsoidal_height);
```

4) Obtain the equivalent `Coord3D` object for the Range 2 SRF by using the `changeCoordinate3D` method of the Range 2 SRF object.

```
Coord3D* range2_coordinate =
```

```
    Range2_SRF->createCoordinate3D(0.0, 0.0, 0.0);

SRM_Coordinate_Valid_Region region = Range2_SRF->changeCoordinate3DSRF(
    *geodetic_coordinate,
    *range2_coordinate);

SRM_Long_Float aircraft_range2_x = 0.0, aircraft_range2_y = 0.0,
    aircraft_range2_z = 0.0;

Range2_SRF->getCoordinate3DValues(*range2_coordinate,
    aircraft_range2_x,
    aircraft_range2_y,
    aircraft_range2_z);
```

6 Orientation

This section addresses the transformation of orientation information from one spatial reference frame to another.

6.1 Concept

Orientation refers to the relationship between the axes of one spatial reference frame, typically the coordinate axes of a rigid body, such as an aircraft, and the axes of a second spatial reference frame. The second SRF is typically referred to as the "world" SRF. Examples include the orientation of a tank SRF with respect to a test/training range (world) SRF, and the orientation of a test/training range SRF with respect to a geodetic (world) SRF. Since a rigid body may be rotated around the point that represents its position (usually its center of mass), the position alone is not enough to completely describe the static state of a rigid body. Orientation is usually symbolized as $\boldsymbol{\Omega}$ (upper case omega).

In general, an orientation describes the relationship between the coordinate axes of two spatial reference frames. In this sense, an orientation can be considered to be an operator that allows vectors, such as velocity vectors, to be transformed from one spatial reference frame to another. Multiple orientations can be composed to form a single compound orientation. For example, if orientation $\boldsymbol{\Omega}_{AB}$ relates the axes of SRF A with the axes of SRF B, and orientation $\boldsymbol{\Omega}_{BC}$ relates the axes of SRF B with the axes of SRF C, then orientation $\boldsymbol{\Omega}_{AC}$ can be composed from orientation $\boldsymbol{\Omega}_{AB}$ and orientation $\boldsymbol{\Omega}_{BC}$ to relate the axes of SRF A with the axes of SRF C.

6.2 Representations

An orientation specifies how to transform the coordinate axes of one spatial reference frame to match the coordinate axes of another spatial reference frame. There are many ways to represent the orientation of a rigid body. The SRM supports five representations:

1. as a rotation angle with respect to a vector (axis-angle), or
2. as a sequence of three rotations around the principal axes (Euler angles, in ***z-x-z*** order: (spin, nutation, and precession), or
3. as a sequence of three rotations around the principal axes (Tait-Bryan angles, in ***x-y-z*** order: roll, pitch, and yaw, or
4. as a 3x3 rotation matrix, or
5. as a quaternion.

6.2.1 Axis-Angle Representation

The axis-angle representation of an orientation consists of a unit vector $\boldsymbol{n}$ (with components n_1, n_2, and n_3) and a rotation angle θ. As shown in Figure 6-1, this represents a rotation of the world coordinate axes through the angle θ about the axis defined by $\boldsymbol{n}$. This rotation, indicated by the red arrows, relates the world coordinate axes (x, y, z), shown in green, with the aircraft body axes (x', y', z'), shown in blue. The green-tinted plane in the figure is parallel to the xy-plane of the world coordinate system, while the blue-tinted plane is the aircraft body $x'y'$-plane. The rotation direction is determined by the right hand rule, i.e., if the right hand grasps the vector, with the thumb pointing in the direction of the vector, the fingers curl around the vector in the direction of the rotation angle θ.

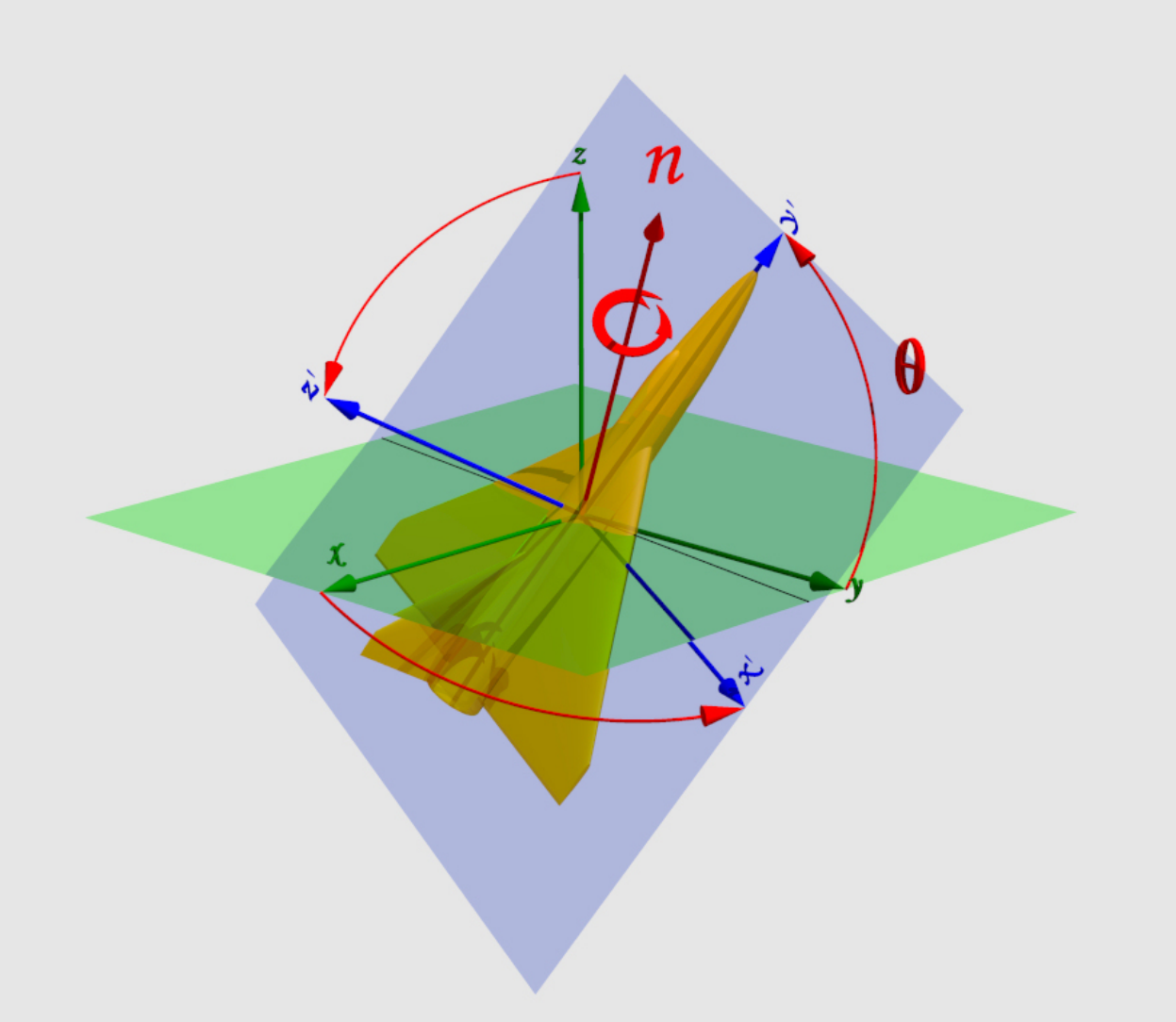

Figure 6-1. Axis-Angle Representation of Orientation

The data type `SRM_Axis_Angle_Params` specifies the parameters that allow an Orientation object to be instantiated using an axis-angle representation. It consists of an axis, specified by a 3D vector, and a rotation angle about that axis. The vector is expressed in terms of its three components in the world SRF. The rotation angle is given in radians.

```
typedef struct
{
    Vector_3D axis;
    SRM_Long_Float angle; /* radians */
} SRM_Axis_Angle_Params;
```

To create an `Orientation` object using the axis-angle representation:

```
// Axis-Angle Representation
SRM_Long_Float n1 = 1.0, n2 = 0.0, n3 = 0.0, theta = 0.123;
SRM_Axis_Angle_Params my_axis_angle_params;
```

```
my_axis_angle_params.axis.array[0] = n1;
my_axis_angle_params.axis.array[1] = n2;
my_axis_angle_params.axis.array[2] = n3;
my_axis_angle_params.angle = theta;

OrientationAxisAngle my_orientation(my_axis_angle_params);
```

The method `getAxisAngle` returns the representation of an `Orientation` object in axis-angle form. It is invoked as follows.

```
my_axis_angle_params = my_orientation.getAxisAngle();
```

6.2.2 Euler Angle Z-X-Z Representation

Euler angles specify an orientation in terms of three consecutive rotations about the principal coordinate system axes. There are twelve distinct ways to select such a sequence of rotations (for right-handed axes). Each of these orderings is called an Euler angle convention. Unfortunately, in the broader community, there is little agreement on how to identify these conventions.

The SRM supports the Euler angle convention identified as the ***z-x-z*** convention. This is also known as the 3-1-3 convention, or the ***x***-convention. (The SRM also supports the Tait-Bryan angle representation, which is another widely used Euler angle convention. See 6.2.3.) As shown in Figure 6-2, this involves a sequence of three rotations that relate the world coordinate axes, shown in green, with the aircraft body coordinate axes, shown in blue. The green tinted plane in the figure is parallel to the world reference system *xy*-plane, while the blue-tinted plane is the aircraft body *x'''y'''*-plane.

The first rotation, $\mathbf{\Omega}_z(\alpha)$, is about the *z*-axis, through angle α. This yields the *x'* and *y'* axes, shown in yellow in Figure 6-2, while the *z* axis, shown in green, remains unchanged.

The second rotation, $\mathbf{\Omega}_x(\beta)$, is about the (original) *x*-axis, through angle β. This yields the *x''*, *y''*, and *z''* axes, shown in orange in Figure 6-2.

The third rotation, $\mathbf{\Omega}_z(\gamma)$, is again about the (original) *z*-axis, through angle γ. This yields the *x'''*, *y'''*, and *z'''* axes, shown in blue in Figure 6-2, which are aligned with the aircraft body axes.

The red arrows in Figure 6-2 show how the *x*, *y* and *z* axes, shown in green, are progressively transformed by each of these rotations to become first the *x'*, *y'* and *z'* axes, shown in yellow, then the *x''*, *y''* and *z''* axes, shown in orange, and finally the *x'''*, *y'''* and *z'''* axes, shown in blue.

In some contexts, α is called the spin angle, β is called the nutation angle, and γ is called the precession angle.

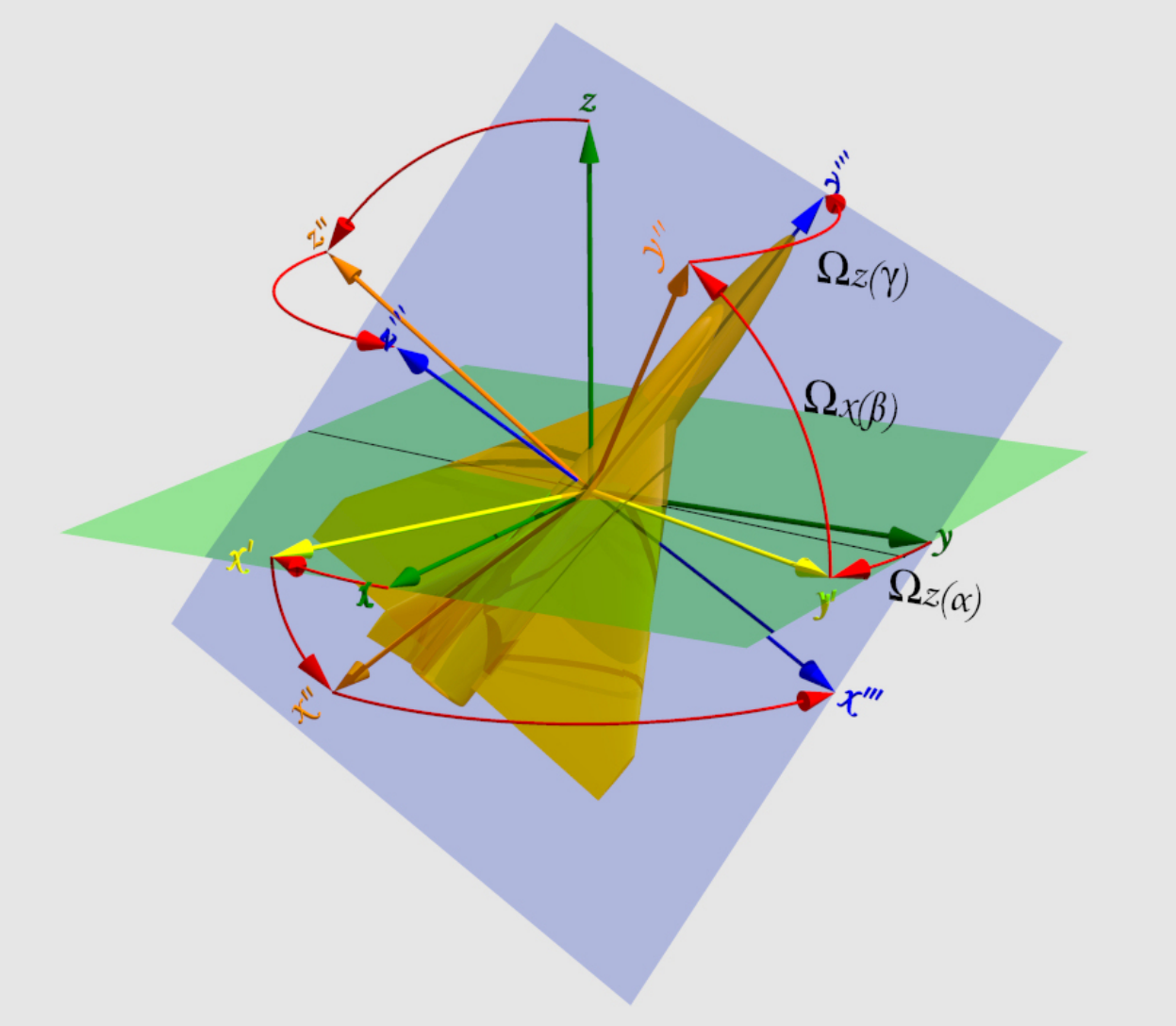

Figure 6-2. Euler Angle Z-X-Z Representation of Orientation

This data type `SRM_Euler_Angles_ZXZ_Params` specifies the parameters that allow an Orientation object to be instantiated using an Euler angles ZXZ representation. It consists of three rotation angles in radians.

```
typedef struct
{
    SRM_Long_Float spin; ; /* radians */
    SRM_Long_Float nutation; ; /* radians */
    SRM_Long_Float precession; ; /* radians */
} SRM_Euler_Angles_ZXZ_Params;
```

To create an `Orientation` object using the Euler angle ***z-x-z*** representation:

```
// Euler Angle Z-X-Z Representation
SRM_Long_Float alpha = 1.0, beta = 0.0, gamma = 0.123;
SRM_Euler_Angles_ZXZ_Params my_euler_angle_params;
```

```
my_euler_angle_params.spin = alpha;
my_euler_angle_params.nutation = beta;
my_euler_angle_params.precession = gamma;

OrientationEulerAnglesZXZ my_orientation(my_euler_angle_params);
```

The method `getEulerAnglesZXZ` returns the representation of an `Orientation` object in Euler angles ZXZ form. It is invoked as follows.

```
my_euler_angle_params = my_orientation.getEulerAnglesZXZ();
```

6.2.3 Tait-Bryan Angle Representation

Another widely used Euler angle convention is the ***x-y-z*** convention.[5] Euler angles in this convention are called Tait-Bryan angles. They are also sometimes called Cardano angles, or nautical angles. As shown in Figure 6-3, this involves a sequence of three rotations that relate the world coordinate axes, shown in green, with the aircraft body coordinate axes, shown in blue. The green tinted plane in the figure is parallel to the world reference system *xy*-plane, while the blue-tinted plane is the aircraft body *x'''y'''*-plane.

The first rotation, $\mathbf{\Omega}_x(\varphi)$, is about the *x*-axis, through angle φ. This gives the *y'*, and *z'* axes, shown in yellow in Figure 6-3, while the *x* axis remains unchanged.

The second rotation, $\mathbf{\Omega}_y(\theta)$, is about the (original) *y*-axis, through angle θ. This gives the *x''*, *y''*, and *z''* axes, shown in orange in Figure 6-3.

The third rotation, $\mathbf{\Omega}_z(\psi)$, is about the (original) *z*-axis, through angle ψ. This gives the *x'''*, *y'''*, and *z'''* axes, shown in blue in Figure 6-3, which are aligned with the aircraft body axes.

The red arrows in Figure 6-3 show how the *x*, *y*, and *z* axes, shown in green, are progressively transformed by each of these rotations to become: first the *x'*, *y'*, and *z'* axes, shown in yellow; then the *x''*, *y''*, and *z''* axis, shown in orange; and finally the *x'''*, *y'''*, and *z'''* axis, shown in blue.

In some contexts, φ is called the roll (or bank or tilt) angle, θ is called the pitch angle, and ψ is called the yaw (or heading or azimuth) angle. Figure 6-3 is consistent with these terms as used with an East-North-Up (ENU) axis convention.

This data type `SRM_Tait_Bryan_Angles_Params` specifies the parameters that allow an `Orientation` object to be instantiated using a Tait-Bryan angles representation. It consists of three rotation angles in radians.

[5] This convention is used to specify orientation in the DIS standard.

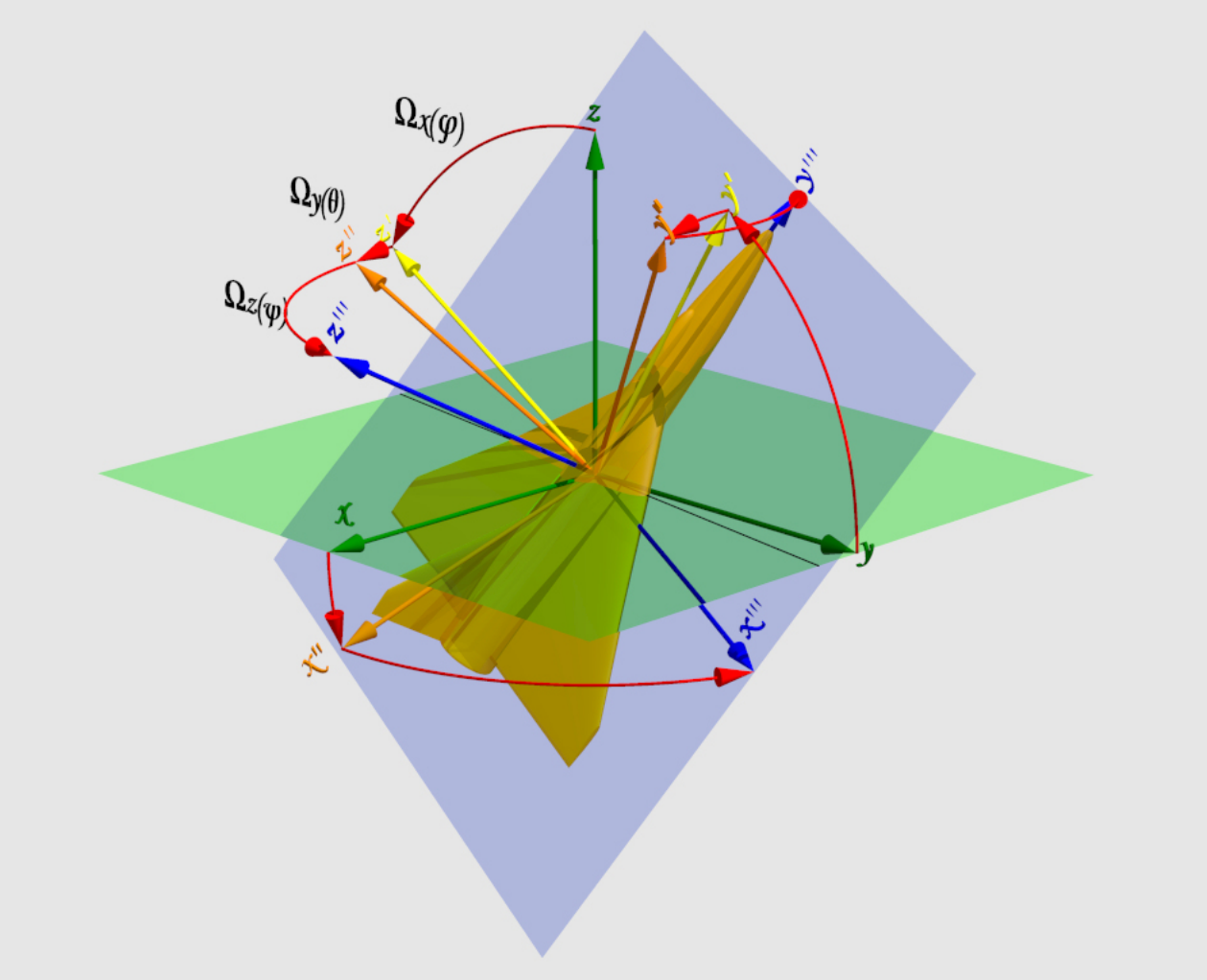

Figure 6-3. Tait-Bryan Angle Representation of Orientation

```
typedef struct
{
    SRM_Long_Float roll; /* radians */
    SRM_Long_Float pitch; /* radians */
    SRM_Long_Float yaw; /* radians */
} SRM_Tait_Bryan_Angles_Params;
```

To create an `Orientation` object using the Tait-Bryan angle representation:

```
// Tait-Bryan Angle Representation
SRM_Long_Float psi = 1.0, theta = 0.0, phi = 0.123;
SRM_Tait_Bryan_Angles_Params my_tait_bryan_angle_params;

my_tait_bryan_angle_params.roll = psi;
my_tait_bryan_angle_params.pitch = theta;
my_tait_bryan_angle_params.yaw = phi;

OrientationTaitBryanAngles my_orientation(my_tait_bryan_angle_params);
```

The method `getTaitBryanAngles` returns the representation of an `Orientation` object in Tait-Bryan angles form. It is invoked as follows.

```
my_tait_bryan_angle_params = my_orientation.getTaitBryanAngles();
```

6.2.4 3x3 Rotation Matrix Representation

The orientation of a rigid body can also be represented in the form of a 3x3 rotation matrix. To represent the Euler angle ***z-x-z*** convention discussed in section 6.2.2, the equivalent 3x3 rotation matrix representation is:

$$\boldsymbol{\Omega}_z(\alpha)\boldsymbol{\Omega}_x(\beta)\boldsymbol{\Omega}_z(\gamma) =$$

$$\begin{pmatrix} \cos\alpha\cos\gamma - \cos\beta\sin\alpha\sin\gamma & \cos\beta\sin\alpha\cos\gamma + \cos\alpha\sin\gamma & \sin\beta\sin\alpha \\ -\sin\alpha\cos\gamma - \cos\beta\cos\alpha\sin\gamma & \cos\beta\cos\alpha\cos\gamma - \sin\alpha\sin\gamma & \sin\beta\cos\alpha \\ \sin\beta\sin\gamma & -\sin\beta\cos\gamma & \cos\beta \end{pmatrix}$$

This data type `SRM_Matrix_3x3` specifies the parameters that allow an Orientation object to be instantiated using a 3x3 rotation matrix representation. It consists of the nine matrix elements.

```
typedef struct
{
    SRM_Long_Float array[3][3];
} SRM_Matrix_3x3;
```

To create an `Orientation` object using the 3x3 rotation matrix representation:

```
// 3x3 Rotation Matrix Representation
SRM_Long_Float alpha = 1.0, beta = 0.0, gamma = 0.123;
SRM_Long_Float a11, a12, a13, a21, a22, a23, a31, a32, a33;

a11 = cos(alpha)*cos(gamma) - cos(beta)*sin(alpha)*sin(gamma);
a12 = cos(beta)*sin(alpha)*cos(gamma) + cos(alpha*sin(gamma);
a13 = sin(beta)*sin(alpha);
a21 = -sin(alpha)*cos(gamma) - cos(beta)*cos(alpha)*sin(gamma);
a22 = cos(beta)*cos(alpha)*cos(gamma) - sin(alpha)*sin(gamma);
a23 = sin(beta)*cos(gamma);
a31 = sin(beta)*sin(gamma);
a32 = -sin(beta)*cos(gamma);
a33 = cos(beta);

SRM_Matrix_3x3 my_matrix_3x3;

my_matrix_3x3.array[0][0] = a11;
my_matrix_3x3.array[0][1] = a12;
my_matrix_3x3.array[0][2] = a13;
my_matrix_3x3.array[1][0] = a21;
my_matrix_3x3.array[1][1] = a22;
my_matrix_3x3.array[1][2] = a23;
my_matrix_3x3.array[2][0] = a31;
my_matrix_3x3.array[2][1] = a32;
my_matrix_3x3.array[2][2] = a33;
```

```
OrientationMatrix my_orientation(my_matrix_3x3);
```

Of course, given the three angles, α, β, and γ, it would be much simpler to create the `Orientation` object using the Euler angles ZXZ representation, as shown in Section 6.2.2. In practice, an `Orientation` object would only be created using the 3x3 rotation matrix representation when the matrix elements are readily available.

For example, to convert between the East-North-Up (ENU) axis convention shown in Figure 6-3 and the North-East-Down (NED) axis convention used by the aeronautical community, an `Orientation` object could be created from the matrix:

$$\boldsymbol{\Omega}_{NED \to ENU} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

```
SRM_Matrix_3x3 ned_enu_matrix_3x3;
ned_enu_matrix_3x3.array[0][0] = 0.0;
ned_enu_matrix_3x3.array[0][1] = 1.0;
ned_enu_matrix_3x3.array[0][2] = 0.0;
ned_enu_matrix_3x3.array[1][0] = 1.0;
ned_enu_matrix_3x3.array[1][1] = 0.0;
ned_enu_matrix_3x3.array[1][2] = 0.0;
ned_enu_matrix_3x3.array[2][0] = 0.0;
ned_enu_matrix_3x3.array[2][1] = 0.0;
ned_enu_matrix_3x3.array[2][2] = -1.0;
}
OrientationMatrix orientation_ned_enu =
              new OrientationMatrix (ned_enu_matrix_3x3);
```

The method `getMatrix3x3` returns the representation of an `Orientation` object in 3x3 rotation matrix form. It is invoked as follows.

```
my_matrix_3x3 = my_orientation.getMatrix3x3();
```

6.2.5 Quaternion Representation

The word "quaternion" means "a set of four." Quaternions are elements of a 4-dimensional vector space. They were first described by the Irish mathematician Sir William Rowan Hamilton in 1843 and applied to mechanics in three-dimensional space. From a purely geometric point of view, a quaternion may be regarded as the quotient of two vectors, or, equivalently, as the operator that transforms one vector into another. Due to certain compactness, efficiency, and stability advantages over matrices, quaternions have found their way into applications in computer graphics, robotics, global navigation, and the orbital mechanics of satellites.

In analogy to complex numbers, quaternion axes $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$, are defined with the following relationships: $\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$. A quaternion $\boldsymbol{q}$ is denoted as $\boldsymbol{q} = e_0 + e_1\boldsymbol{i} + e_2\boldsymbol{j} + e_3\boldsymbol{k}$. This is known as the Hamilton form. The first term e_0 is called the "real" (or "scalar") part of $\boldsymbol{q}$

and $e_1\boldsymbol{i} + e_2\boldsymbol{j} + e_3\boldsymbol{k}$ is called the "imaginary" (or "vector") part of $\boldsymbol{q}$. The SRM uses a convention known as the 4-tuple form, which is simply the 4-tuple of numbers $\boldsymbol{q} = (e_0, e_1, e_2, e_3)$.

The data type `SRM_Quaternion_Params` specifies the parameters that allow an Orientation object to be instantiated using a quaternion representation. It consists of a 4-tuple of numbers, the scalar part and the three vector parts. The parameter values must meet the constraint:
$e_0^2 + e_1^2 + e_2^2 + e_3^2 = 1$

```
typedef struct
{
    SRM_Long_Float e0; /* scalar component */
    SRM_Long_Float e1; /* i component */
    SRM_Long_Float e2; /* j component */
    SRM_Long_Float e3; /* k component */
} SRM_Quaternion_Params;
```

To create an `Orientation` object using the quaternion representation:

```
// Quaternion Representation
SRM_Long_Float e_0 = 0.97098796, e_1 = -0.10497177, e_2 = 0.18746797,
  e_3 = 0.10497177;
SRM_Quaternion_Params my_quaternion_params;

my_quaternion_params.e0 = e_0;
my_quaternion_params.e1 = e_1;
my_quaternion_params.e2 = e_2;
my_quaternion_params.e3 = e_3;

OrientationQuaternion my_orientation(my_quaternion_params);
```

The method `getQuaternion` returns the representation of an `Orientation` object in quaternion form. It is invoked as follows.

```
my_quaternion_params = my_orientation.getQuaternion();
```

6.2.6 Orientation Representation Access

Once it has been defined, an `Orientation` object can return its value in any of the supported representations described above. This is accomplished using a collection of access methods that are supported by all `Orientation` objects.

Given an `Orientation` object called `my_orientation`, to obtain its axis-angle representation:

```
SRM_Axis_Angle_Params my_axis_angle_params =
    my_orientation.getAxisAngle();
```

To obtain its Euler angle ***z-x-z*** representation:

```
SRM_Euler_Angles_ZXZ_Params my_euler_angle_params =
    my_orientation.getEulerAnglesZXZ();
```

To obtain its Tait-Bryan angle representation:

```
SRM_Tait_Bryan_Angles_Params my_tait_bryan_angle_params =
    my_orientation.getTaitBryanAngles();
```

To obtain its 3x3 rotation matrix representation:

```
SRM_Matrix_3x3 my_matrix_3x3 = my_orientation.getMatrix3x3();
```

And finally, to obtain its quaternion representation:

```
SRM_Quaternion_Params my_quaternion_params =
    my_orientation.getQuaternion();
```

6.3 Transformation procedure

The general procedure for transforming an orientation from one SRF to another is:
1) Create the source SRF object, as shown in Section 4.
2) Create the target SRF object, as shown in Section 4.
3) Create the source `Orientation` object that relates the rigid body axes to the source SRF.
4) Obtain the target `Orientation` object using the `transformOrientation` (or `transformOrientationCommonOrigin`) method of the target SRF.
5) Retrieve the desired representation of the target `Orientation` object.

The two variations of the `transformOrientation` method are described below.

6.3.1 Transform Orientation

Given an orientation with respect to a local tangent frame (LTF_S) associated with a reference location in the source SRF, the method `transformOrientation` computes the orientation with respect to the local tangent frame (LTF_T) associated with the specified reference location in the target SRF. The output orientation is computed by composing the orientation of LTF_S with respect to LTF_T with the source orientation. The invoking SRF is the target SRF.

This method takes 3 input parameters:
1) source reference location (a coordinate in the source SRF) where the origin of LTF_S is located. (In the C++ and Java implementations, the source SRF is implied by the source reference location.)
2) source orientation of some linear reference frame with respect to LTF_S.
3) target reference location (a coordinate in this SRF, the target SRF) where the origin of LTF_T is located.

This method computes 1 output parameter:
1) target orientation with respect to LTF_T.

It is invoked as follows:

```
target_SRF->transformOrientation (
        source_ref_coord,  /* input, Coord3D */
        source_orientation,  /* input, Orientation */
        target_ref_coord,  /* input, Coord3D */
```

```
        target_orientation);  /* output, Orientation */
```

6.3.2 Transform Orientation with Common Origin

Given an orientation with respect to a local tangent frame (LTF_S) associated with a reference location in the source SRF, the method `transformOrientationCommonOrigin` computes the orientation with respect to the local tangent frame (LTF_T) associated with the specified reference location in the target SRF. LTF_S and LTF_T have a common origin. The output orientation is computed by composing the orientation of LTF_S with respect to LTF_T with the input orientation. The invoking SRF is the target SRF.

This method takes 2 input parameters:
1) source reference location (a coordinate in the source SRF) where the origin of LTF_S is located.
2) source orientation of some linear reference frame with respect to LTF_S.

This method computes 2 output parameters:
1) coordinate of the common reference location in the target SRF, computed from the source reference location coordinate.
2) target orientation with respect to LTF_T.

It is invoked as follows:

```
target_SRF->transformOrientationCommonOrigin (
        source_ref_coord,  /* input, Coord3D */
        source_orientation,  /* input, Orientation */
        target_ref_coord,  /* output, Coord3D */
        target_orientation);  /* output, Orientation */
```

6.4 Examples

The following examples show how to transform orientations between the various SRFs discussed in Section 4.

6.4.1 Transform Between Range SRFs

Example 1: Transform the orientation of an aircraft in axis-angle form with respect to the Range 1 SRF to the Range 2 SRF, also in axis-angle form. As shown in Figure 6-4, the axes of the two Range SRFs are not parallel. Therefore, the orientation of the aircraft with respect to the Range 1 SRF is different from its orientation with respect to the Range 2 SRF.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
3) Create the source `Orientation` object that relates the aircraft body axes (shown in blue in Figure 6-4) to the axes of the Range 1 SRF (shown in green in Figure 6-4) in axis-angle form. This orientation is shown as $\mathbf{\Omega}_{\text{range1-to-body}}$ in Figure 6-4.

```
// From Range 1 (Axis-Angle) to Range 2 (Axis-Angle)
SRM_Axis_Angle_Params aircraft_range1_axis_angle_params;
SRM_Long_Float n1 = -0.43301270, n2 = 0.75, n3 = 0.5;

aircraft_range1_axis_angle_params.axis.array[0] = n1;
aircraft_range1_axis_angle_params.axis.array[1] = n2;
```

```
aircraft_range1_axis_angle_params.axis.array[2] = n3;
aircraft_range1_axis_angle_params.angle = 30.0 * degreesToRadians;
```

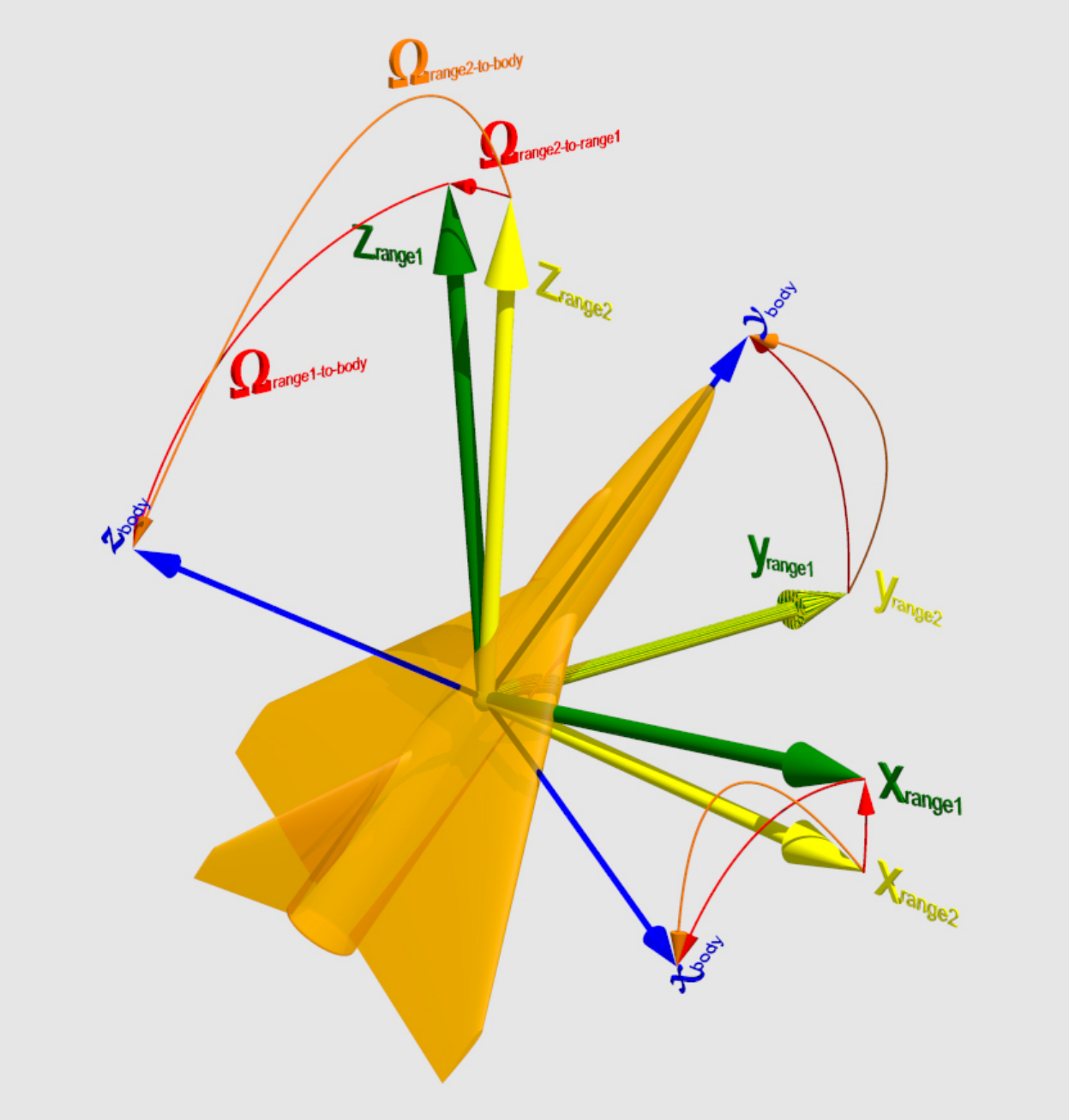

Figure 6-4. Orientation Transformation from Range 1 to Range 2

```
OrientationAxisAngle aircraft_range1_orientation(
    aircraft_range1_axis_angle_params);
```

4) Obtain the Range 2 `Orientation` object using the `transformOrientation` (or `transformOrientationCommonOrigin`) method of the Range 2 SRF. This method first computes the `Orientation` object that relates the Range 1 SRF to the Range 2 SRF at the specified reference coordinates, shown as $\boldsymbol{\Omega}_{\text{range2-to-range1}}$ in Figure 6-4. This is then composed

with the Range 1 `Orientation` object to create the Range 2 `Orientation` object. This method requires a reference coordinate for each of the two SRFs. However, because the Range SRFs are both linear, any convenient locations can be chosen. Suppose the current Range 1 coordinate of the aircraft is chosen. The `transformOrientationCommonOrigin` method can then be used to transform both the position and the orientation of the aircraft to the Range 2 SRF at the same time.

```
SRM_Long_Float aircraft_range1_x = 5000.0,
    aircraft_range1_y = 100000.0,
    aircraft_range1_z = 5000.0; // meters

Coord3D* aircraft_range1_coord = Range1_SRF->createCoordinate3D(
    aircraft_range1_x,
    aircraft_range1_y,
    aircraft_range1_z);

Coord3D* aircraft_range2_coord = Range2_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

OrientationTaitBryanAngles aircraft_range2_orientation;

Range2_SRF->transformOrientationCommonOrigin(
    *aircraft_range1_coord,
    aircraft_range1_orientation,
    *aircraft_range2_coord,
    aircraft_range2_orientation);

SRM_Long_Float aircraft_range2_x, aircraft_range2_y, aircraft_range2_z;

Range2_SRF->getCoordinate3DValues(*aircraft_range2_coord,
    aircraft_range2_x,
    aircraft_range2_y,
    aircraft_range2_z);
```

5) Retrieve the axis-angle representation of the Range 2 `Orientation` object.

```
SRM_Axis_Angle_Params aircraft_range2_axis_angle_params =
    aircraft_range2_orientation.getAxisAngle();

SRM_Long_Float aircraft_range2_axis_n1, aircraft_range2_axis_n2,
    aircraft_range2_axis_n3, aircraft_range2_axis_theta;

aircraft_range2_axis_n1 =
    aircraft_range2_axis_angle_params.axis.array[0];
aircraft_range2_axis_n2 =
    aircraft_range2_axis_angle_params.axis.array[1];
aircraft_range2_axis_n3 =
    aircraft_range2_axis_angle_params.axis.array[2];
aircraft_range2_axis_theta = aircraft_range2_axis_angle_params.angle;
```

6.4.2 Transform From Range to Geocentric

Example 2: Transform the orientation of a tank with respect to the Range 1 SRF in quaternion form to the Geocentric WGS 1984 SRF in Euler angle form.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Geocentric WGS 1984 SRF (see Section 4.1.2).
3) Create the source `Orientation` object that relates the tank body axes to the axes of the Range 1 SRF in quaternion form.

```
// From Range 1 (Quaternion) to Geocentric (Euler Angle)
SRM_Quaternion_Params tank_range1_quaternion_params;

tank_range1_quaternion_params.e0 = 0.96607316; // real
tank_range1_quaternion_params.e1 = -0.11195108; // i
tank_range1_quaternion_params.e2 = 0.19376985; // j
tank_range1_quaternion_params.e3 = 0.12892965; // k

OrientationQuaternion tank_range1_orientation(
    tank_range1_quaternion_params);
```

4) Obtain the Geocentric WGS 1984 `Orientation` object using the `transformOrientation` (or `transformOrientationCommonOrigin`) method of the Geocentric WGS 1984 SRF. This method first computes the `Orientation` object that relates the Range 1 SRF to the Geocentric WGS 1984 SRF at the specified reference coordinates. This is then composed with the Range 1 `Orientation` object to create the Geocentric WGS 1984 `Orientation` object. This method requires a reference coordinate for each of the two SRFs. Because the Range 1 SRF and the Geocentric WGS 1984 SRF are both linear, any convenient locations can be chosen. Suppose the current Range 1 coordinate of the tank is chosen. The `transformOrientationCommonOrigin` method can then be used to transform both the position and the orientation of the tank to the Geocentric WGS 1984 SRF at the same time.

```
SRM_Long_Float tank_range1_x = 2000.0,
    tank_range1_y = 5000.0,
    tank_range1_z = 500.0; // meters

Coord3D* tank_range1_coord = Range1_SRF->createCoordinate3D(
    tank_range1_x,
    tank_range1_y,
    tank_range1_z);

Coord3D* tank_geocentric_coord =
    Geocentric_WGS84_SRF->createCoordinate3D(0.0, 0.0, 0.0);

OrientationEulerAnglesZXZ tank_geocentric_orientation;

Geocentric_WGS84_SRF->transformOrientationCommonOrigin(
    *tank_range1_coord,
    tank_range1_orientation,
    *tank_geocentric_coord,
    tank_geocentric_orientation);

SRM_Long_Float tank_geocentric_x, tank_geocentric_y, tank_geocentric_z;

Geocentric_WGS84_SRF->getCoordinate3DValues(*tank_geocentric_coord,
    tank_geocentric_x,
    tank_geocentric_y,
    tank_geocentric_z);
```

5) Retrieve the Euler angle representation of the Geocentric WGS 1984 `Orientation` object.

```
SRM_Euler_Angles_ZXZ_Params tank_euler_angle_params =
    tank_geocentric_orientation.getEulerAnglesZXZ();

SRM_Long_Float tank_alpha = tank_euler_angle_params.spin;
SRM_Long_Float tank_beta = tank_euler_angle_params.nutation;
SRM_Long_Float tank_gamma = tank_euler_angle_params.precession;
```

6.4.3 Transform From Range to Geodetic

Example 3: Transform the orientation of a tank with respect to the Range 1 SRF in 3x3 matrix form to the Geodetic WGS 1984 SRF in Tait-Bryan angle form.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
3) Create the source `Orientation` object that relates the tank body axes to the axes of the Range 1 SRF in 3x3 matrix form.

```
// From Range 1 (3x3 Matrix) to Geodetic (Tait-Bryan)
SRM_Matrix_3x3 tank_range1_matrix = {
  0. 0.89114923, -0.29350295, 0.34598999,
  0.20648575, 0.94138810, 0.26674346,
 -0.40400085, -0.16626623, 0.89952146 };

OrientationMatrix tank_range1_orientation(tank_range1_matrix);
```

4) Obtain the Geodetic WGS 1984 `Orientation` object, using the `transformOrientation` (or `transformOrientationCommonOrigin`) method of the Geodetic WGS 1984 SRF. This method first computes the `Orientation` object that relates the Range 1 SRF to the Geodetic WGS 1984 SRF at the specified reference coordinates. This is then composed with the Range 1 `Orientation` object to create the Geodetic WGS 1984 `Orientation` object. This requires a reference coordinate for each of the two SRFs. In this case, because the Range 1 SRF is a linear SRF, any convenient location can be chosen for the Range 1 reference coordinate. However, because the Geodetic WGS 1984 SRF is a curvilinear SRF, an appropriate and relevant location should be chosen as the geodetic reference coordinate. Suppose the current Range 1 coordinate of the tank is chosen as the reference coordinate for the Range 1 SRF, and the Geodetic WGS 1984 coordinate of the tank is chosen as the reference coordinate for the Geodetic WGS 1984 SRF. This defines a Local Tangent Space Euclidean SRF with its origin located at the current position of the tank. The `transformOrientationCommonOrigin` method can then be used to transform both the position and the orientation of the tank to the Geodetic WGS 1984 SRF at the same time.

```
SRM_Long_Float tank_range1_x = 2000.0,
    tank_range1_y = 5000.0,
    tank_range1_z = 500.0; // meters

Coord3D* tank_range1_coord = Range1_SRF->createCoordinate3D(
    tank_range1_x,
    tank_range1_y,
    tank_range1_z);
```

```
Coord3D* tank_geodetic_coord = Geodetic_WGS84_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

OrientationTaitBryanAngles tank_geodetic_orientation;

Geodetic_WGS84_SRF->transformOrientationCommonOrigin(
    *tank_range1_coord,
    tank_range1_orientation,
    *tank_geodetic_coord,
    tank_geodetic_orientation);

SRM_Long_Float tank_geodetic_longitude,
    tank_geodetic_latitude,
    tank_geodetic_ellipsoid_height;

Geodetic_WGS84_SRF->getCoordinate3DValues(*tank_geodetic_coord,
    tank_geodetic_longitude,
    tank_geodetic_latitude,
    tank_geodetic_ellipsoid_height);
```

5) Retrieve the Tait-Bryan angle representation of the Geodetic WGS 1984 `Orientation` object.

```
SRM_Tait_Bryan_Angles_Params tank_tait_bryan_angle_params =
    tank_geodetic_orientation.getTaitBryanAngles();

SRM_Long_Float tank_psi = tank_tait_bryan_angle_params.roll;
SRM_Long_Float tank_theta = tank_tait_bryan_angle_params.pitch;
SRM_Long_Float tank_phi = tank_tait_bryan_angle_params.yaw;
```

6.4.4 Transform From Geodetic to Geocentric

Example 4: Transform the orientation of an aircraft with respect to the Geodetic WGS 1984 SRF in Tait-Bryan angle form to the Geocentric WGS 1984 SRF in Euler angle form.

1) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
2) Create the SRF object for the Geocentric WGS 1984 SRF (see Section 4.1.2).
3) Create the Geodetic WGS 1984 `Orientation` object that relates the aircraft body axes to the axes of the Geodetic WGS 1984 SRF in Tait-Bryan angle form.

```
// From Geodetic (Tait-Bryan) to Geocentric (Euler Angle)
SRM_Tait_Bryan_Angles_Params aircraft_geodetic_tait_bryan_params;

aircraft_geodetic_tait_bryan_params.roll = 30.0 * degreesToRadians;
aircraft_geodetic_tait_bryan_params.pitch = 30.0 * degreesToRadians;
aircraft_geodetic_tait_bryan_params.yaw = 120.0 * degreesToRadians;

OrientationTaitBryanAngles aircraft_geodetic_orientation(
    aircraft_geodetic_tait_bryan_params);
```

4) Obtain the Geocentric WGS 1984 `Orientation` object, using the `transformOrientation` method of the Geocentric WGS 1984 SRF. This method first computes the `Orientation` object that relates the Geodetic WGS 1984 SRF to the Geocentric WGS 1984 SRF at the

specified reference coordinates. This is then composed with the Geodetic WGS 1984 `Orientation` object to create the Geocentric WGS 1984 `Orientation` object. This requires a reference coordinate for each of the two SRFs. In this case, because the Geodetic WGS 1984 SRF is a curvilinear SRF, an appropriate and relevant location should be chosen as the geodetic reference coordinate. Suppose the Geodetic WGS 1984 coordinate of the aircraft is chosen as the geodetic reference coordinate. This defines a Local Tangent Space Euclidean SRF with its origin located at the current position of the aircraft. However, because the Geocentric WGS 1984 SRF is a linear SRF, any convenient location can be chosen as the geocentric reference coordinate. Suppose the geocentric origin is chosen.

```
SRM_Long_Float aircraft_longitude = -120.5 * degreesToRadians,
    aircraft_latitude = 33.5 * degreesToRadians,
    aircraft_ellipsoidal_height = 5000; // meters

Coord3D* geodetic_ref_coord = Geodetic_WGS84_SRF->createCoordinate3D(
    aircraft_longitude,
    aircraft_latitude,
    aircraft_ellipsoidal_height);

Coord3D* geocentric_ref_coord = Geocentric_WGS84_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

OrientationEulerAnglesZXZ aircraft_geocentric_orientation;

Geocentric_WGS84_SRF->transformOrientation(
    *geodetic_ref_coord,
    aircraft_geodetic_orientation,
    *geocentric_ref_coord,
    aircraft_geocentric_orientation);
```

5) Retrieve the Euler angle representation of the Geocentric WGS 1984 `Orientation` object.

```
 SRM_Euler_Angles_ZXZ_Params aircraft_euler_angle_params =
    aircraft_geocentric_orientation.getEulerAnglesZXZ();

SRM_Long_Float aircraft_alpha = aircraft_euler_angle_params.spin;
SRM_Long_Float aircraft_beta = aircraft_euler_angle_params.nutation;
SRM_Long_Float aircraft_gamma = aircraft_euler_angle_params.precession;
```

6.4.5 Transform From Geodetic to Range

Example 5: Transform the orientation of an aircraft with respect to the Geodetic WGS 1984 SRF in Tait-Bryan angle form to the Range 2 SRF in quaternion form.

1) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
2) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
3) Create the Geodetic WGS 1984 `Orientation` object that relates the aircraft body axes to the axes of the Geodetic WGS 1984 SRF in Tait-Bryan angle form.

```
// From Geodetic (Tait-Bryan) to Range 2 (Quaternion)
SRM_Tait_Bryan_Angles_Params aircraft_geodetic_tait_bryan_params;

aircraft_geodetic_tait_bryan_params.roll = 30.0 * degreesToRadians;
```

```
aircraft_geodetic_tait_bryan_params.pitch = 30.0 * degreesToRadians;
aircraft_geodetic_tait_bryan_params.yaw = 120.0 * degreesToRadians;

OrientationTaitBryanAngles aircraft_geodetic_orientation(
    aircraft_geodetic_tait_bryan_params);
```

4) Obtain the Range 2 `Orientation` object, using the `TransformOrientation` method of the Range 2 SRF. This method first computes the `Orientation` object that relates the Geodetic WGS 1984 SRF to the Range 2 SRF at the specified reference coordinates. This is then composed with the Geodetic WGS 1984 `Orientation` object to create the Range 2 `Orientation` object. This requires a reference coordinate for each of the two SRFs. In this case, because the Geodetic WGS 1984 SRF is a curvilinear SRF, an appropriate and relevant location should be chosen as the geodetic reference coordinate. Suppose the Geodetic WGS 1984 coordinate of the aircraft is chosen. This defines a Local Tangent Space Euclidean SRF with its origin located at the current position of the aircraft. However, because the Range 2 SRF is a linear SRF, any convenient location can be chosen as the Range 2 reference coordinate. Suppose the Range 2 origin is chosen.

```
SRM_Long_Float aircraft_longitude = -120.5 * degreesToRadians,
    aircraft_latitude = 33.5 * degreesToRadians,
    aircraft_ellipsoidal_height = 5000; // meters

Coord3D* geodetic_ref_coord = Geodetic_WGS84_SRF->createCoordinate3D(
    aircraft_longitude,
    aircraft_latitude,
    aircraft_ellipsoidal_height);

Coord3D* range2_ref_coord = Range2_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

OrientationQuaternion aircraft_range2_orientation;

Range2_SRF->transformOrientation(
    *geodetic_ref_coord,
    aircraft_geodetic_orientation,
    *range2_ref_coord,
    aircraft_range2_orientation);
```

5) Retrieve the quaternion representation of the Range 2 `Orientation` object.

```
SRM_Quaternion_Params aircraft_quaternion_params =
    aircraft_range2_orientation.getQuaternion();

SRM_Long_Float aircraft_e0 = aircraft_quaternion_params.e0;
SRM_Long_Float aircraft_e1 = aircraft_quaternion_params.e1;
SRM_Long_Float aircraft_e2 = aircraft_quaternion_params.e2;
SRM_Long_Float aircraft_e3 = aircraft_quaternion_params.e3;
```

7 Vector Quantities

This section addresses the transformation of vector quantities from one spatial reference frame to another, including linear velocity, angular velocity, linear acceleration, and angular acceleration.

7.1 Concepts

The concepts of linear velocity, angular velocity, linear acceleration, and angular acceleration are briefly reviewed in this subsection.

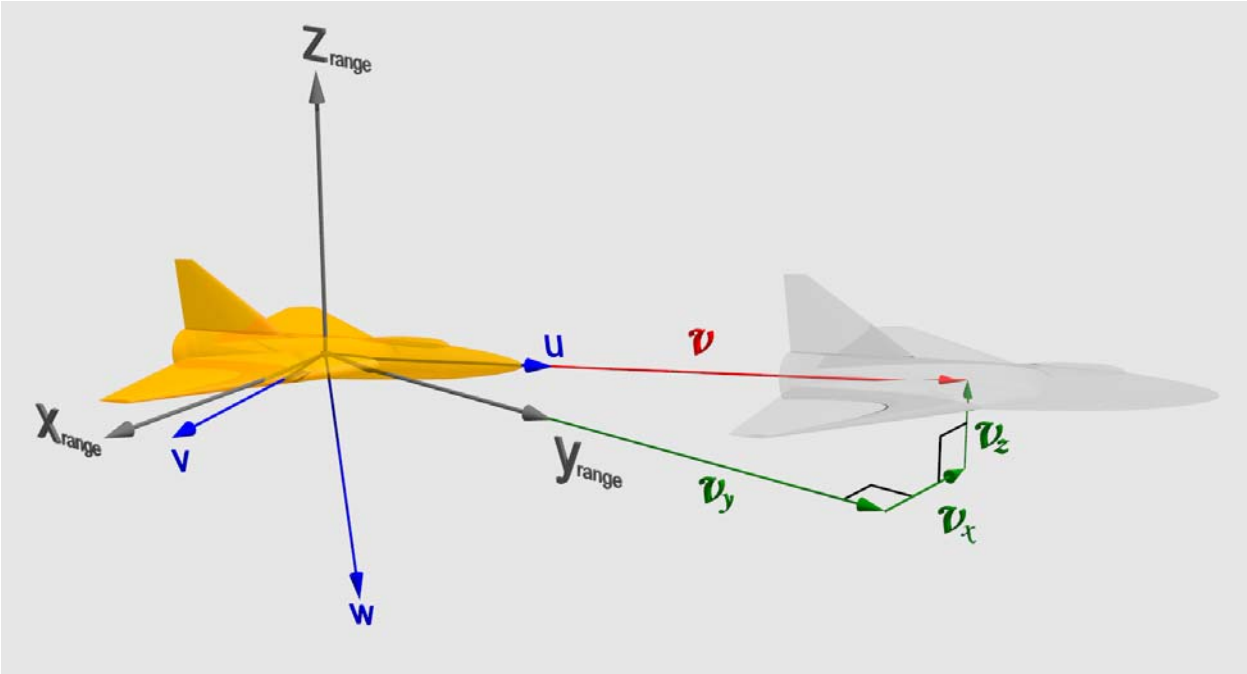

Figure 7-1. Linear Velocity

7.1.1 Linear Velocity

Linear velocity is the time rate of change of displacement (i.e., change of position), or the derivative of displacement with respect to time, of a particle or rigid body. Linear velocity is a vector, which consists of both a direction and a magnitude, i.e., speed. Linear velocity is commonly symbolized as $\boldsymbol{v}$. Figure 7-1 shows the current position of an aircraft, its linear velocity vector (shown in red), and its projected position after a time interval (shown as a "ghost" of the aircraft). The instantaneous linear velocity, which specifies the linear velocity of the particle or rigid body at a specific point in time, is the limit as the length of the interval approaches zero. The units for linear velocity are typically meters per second (m/sec).

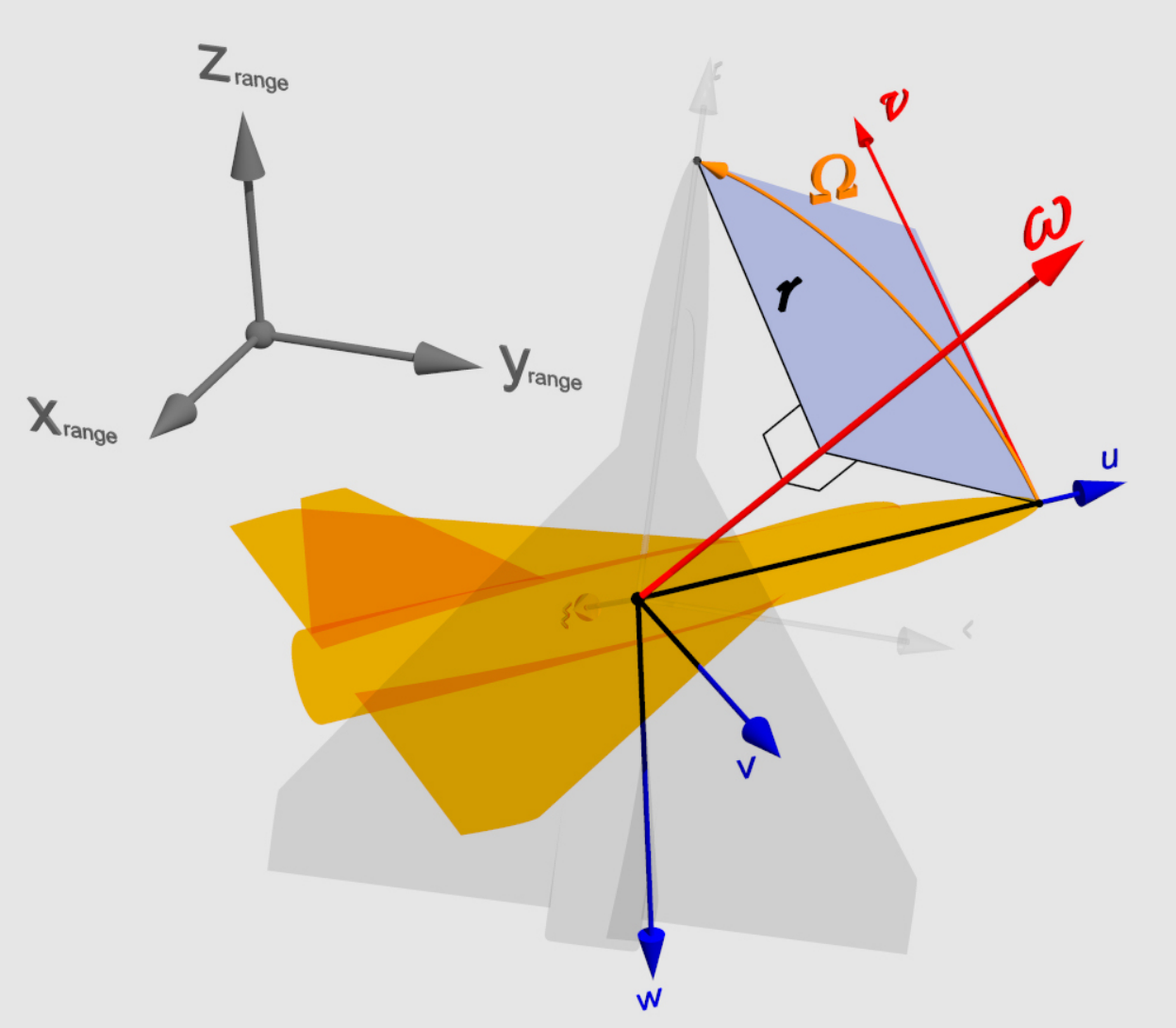

Figure 7-2. Angular Velocity

7.1.2 Angular Velocity

Angular velocity is the time rate of rotation, or the derivative of rotation with respect to time, of a particle or a rigid body about an axis. Angular velocity is a vector, with magnitude equal to the angular speed at which the body is rotating, and which points in the direction of the axis of rotation. The direction of rotation about the axis is specified by the right hand rule. Angular velocity is commonly symbolized as **ω** (lower case omega). Figure 7-2 shows the angular velocity of an aircraft as it rotates about an arbitrary axis, shown in red, over a time interval. The future position of the aircraft is shown in the form of a "ghost" aircraft. The orientation **Ω** specifies the relationship between the current and future aircraft body axes. The tangential velocity vector ***v*** shows the linear velocity of a point on the nose of the aircraft, which is located a distance *r* from the axis of rotation. The orange arc shows the actual path of this point, within a plane perpendicular to the axis of rotation, shown in blue, as the aircraft rotates. The units for angular velocity are typically radians per second (rad/sec).

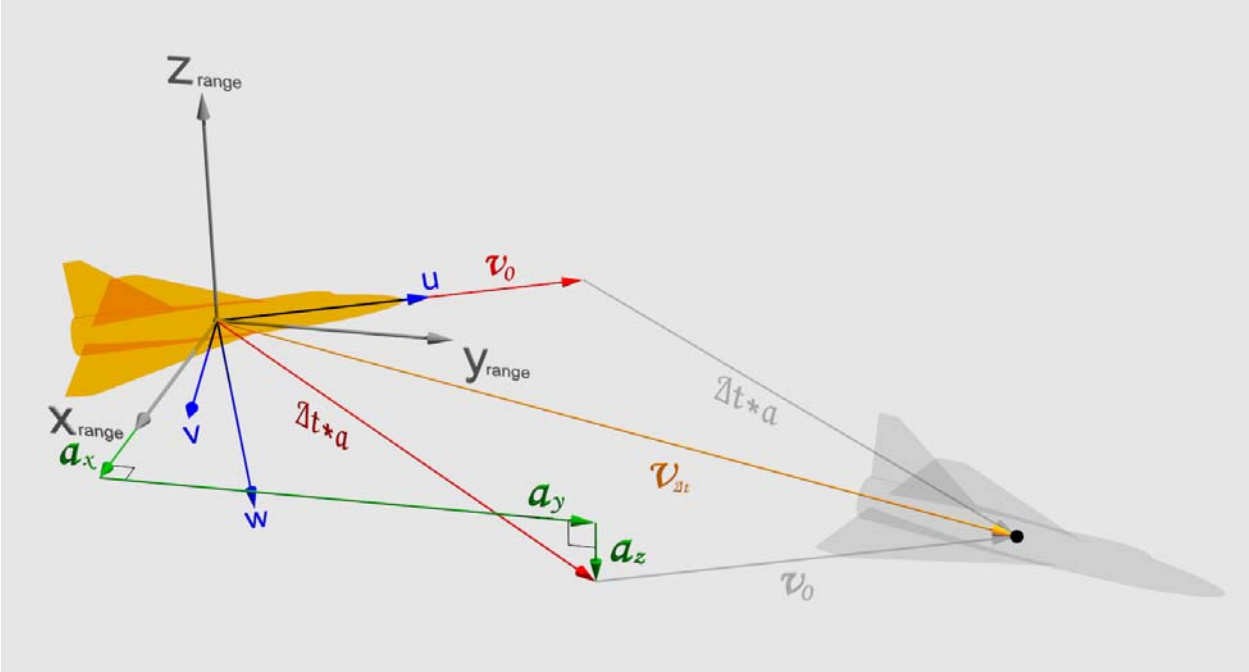

Figure 7-3. Linear Acceleration

7.1.3 Linear Acceleration

Linear acceleration is the time rate of change of linear velocity, or the derivative of linear velocity with respect to time, of a particle or rigid body. Linear acceleration is a vector. It is commonly symbolized as ***a***. Figure 7-3 shows the change in the linear velocity of an aircraft, over the course of a brief time interval, Δt. The vector $\boldsymbol{v}_0$ is the current linear velocity vector. The vector ***a*** is the current linear acceleration vector. Its components are also shown, in green. The future position of the aircraft is shown as a "ghost" aircraft. The vector $\boldsymbol{v}_{\Delta t}$ is an approximation of its future linear velocity vector, given by $\boldsymbol{v}_{\Delta t} \cong \boldsymbol{v}_0 + (\Delta t * \boldsymbol{a})$. The instantaneous linear acceleration, which specifies the linear acceleration at a specific point in time, is the limit as the length of the time interval approaches zero. The units for linear acceleration are typically meters per second squared (m/sec^2).

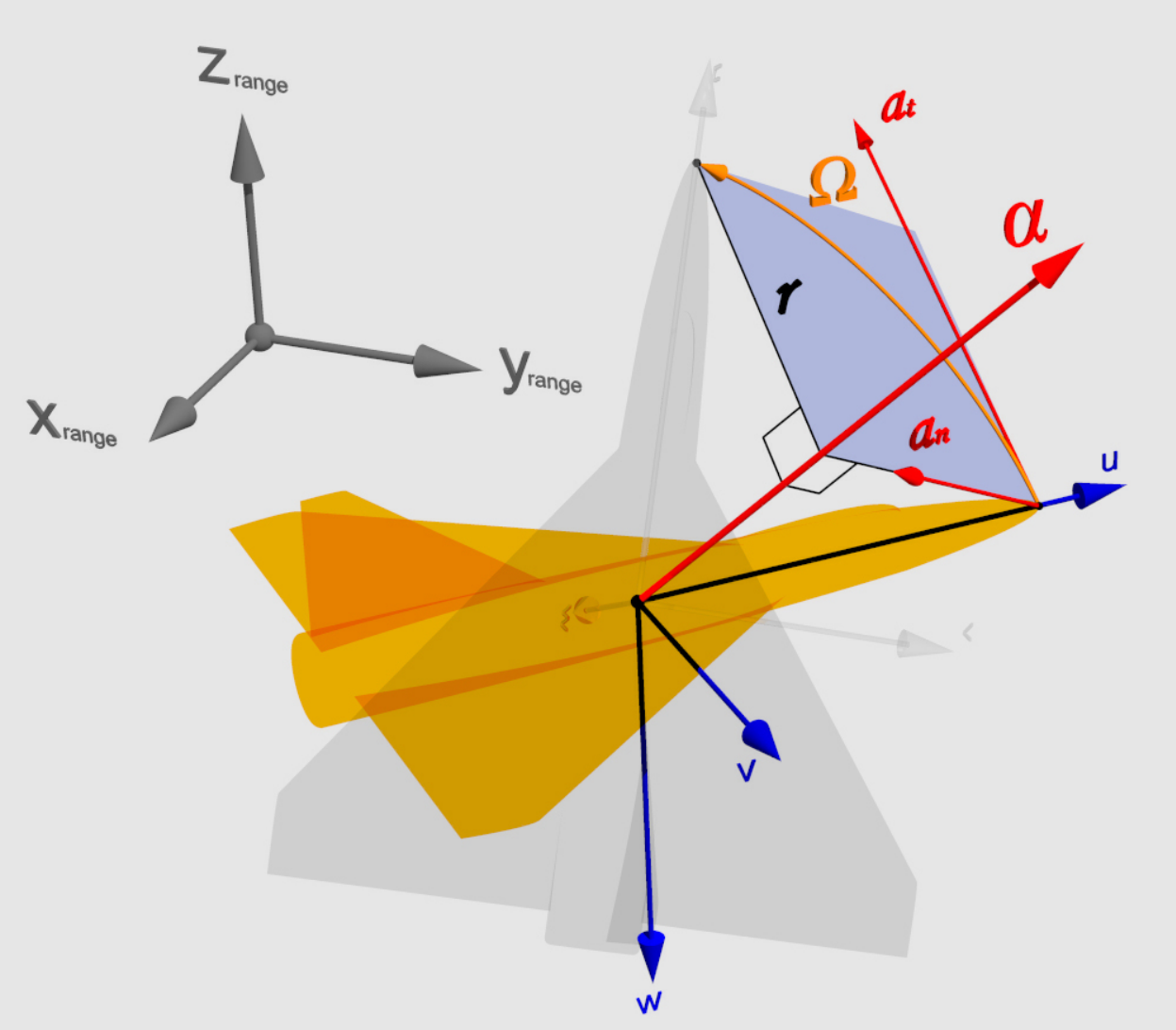

Figure 7-4. Angular Acceleration

7.1.4 Angular Acceleration

Angular acceleration is the time rate of change of angular velocity, or the derivative of angular velocity with respect to time, of a particle or rigid body. Angular acceleration is a vector. It is commonly symbolized as $\boldsymbol{\alpha}$ (lower case alpha)[6]. Figure 7-4 shows the change in the angular velocity of an aircraft, as it rotates about an arbitrary axis, over the course of a time interval. The future position of the aircraft is shown in the form of a "ghost" aircraft. The orientation $\boldsymbol{\Omega}$ specifies the relationship between the current and future aircraft body axes. The tangential ($\boldsymbol{a_t}$) and normal ($\boldsymbol{a_n}$) components of the angular acceleration show how the velocity of a point on the nose of the aircraft, located a distance *r* from the axis of rotation, changes over time. The orange arc shows the actual path of this point, within a plane perpendicular to the axis of rotation, shown in blue, as the aircraft rotates. The units for angular acceleration are typically radians per second squared (rad/sec^2).

[6] The angular acceleration vector $\boldsymbol{\alpha}$ should not be confused with the azimuth angle α used in section 4.2.1.

7.2 Representation

Several quantities associated with a particle or rigid body, including linear velocity, angular velocity, linear acceleration, and angular acceleration, can be represented as 3D vectors. In the SRM API, such 3D vectors are instantiated using the data type `Vector_3D_Params`:

```
typedef struct
{
    SRM_Long_Float array[3];
} SRM_Vector_3D;
```

The three components of this vector represent the three components of the linear velocity, angular velocity, linear acceleration, or angular acceleration along each of the coordinate axes of the appropriate SRF, in order.

The vector transformation operations in the SRM do not require the units of a given vector quantity, since the units are not affected by any vector transformation operations.

7.3 Transformation Procedure

The general procedure for transforming one or more vector quantities from one SRF to another is:

1) Create the source SRF object, as shown in Section 4.
2) Create the target SRF object, as shown in Section 4.
3) Transform the vector(s) from the source SRF to the target SRF using the `transformVector` method (or one of its alternative forms) of the target SRF.

The four variations of the `transformVector` method are described below.

7.3.1 Transform Vector

Given a vector in the local tangent frame (LTF_S) associated with a reference location in the source SRF, the method `transformVector` computes the vector in the local tangent frame (LTF_T) associated with the specified reference location in the target SRF. The output vector is computed by applying the orientation of LTF_S with respect to LTF_T to the source vector. The invoking SRF is the target SRF.

This method takes 3 input parameters:

1) source reference location (a coordinate in the source SRF) where the origin of LTF_S is located. (In the C++ and Java implementations, the source SRF is implied by the source reference location.)
2) source vector in LTF_S.
3) target reference location (a coordinate in this SRF, the target SRF) where the origin of LTF_T is located.

This method computes 1 output parameter:

1) target vector in LTF_T.

It is invoked as follows:

```
target_SRF->transformVector (
        source_ref_coord,  /* input, Coord3D */
        source_vector,  /* input, Vector_3D */
        target_ref_coord,  /* input, Coord3D */
        target_vector);  /* output, Vector_3D */
```

7.3.2 Transform Vector with Common Origin

Given a vector in the local tangent frame (LTF_S) associated with a reference location in the source SRF, the method `transformVectorCommonOrigin` computes the vector in the local tangent frame (LTF_T) associated with the specified reference location in the target SRF. LTF_S and LTF_T have a common origin. The output vector is computed by applying the orientation of LTF_S with respect to LTF_T to the source vector. The invoking SRF is the target SRF.

This method takes 2 input parameters:
1) source reference location (a coordinate in the source SRF) where the origin of LTF_S is located. (In the C++ and Java implementations, the source SRF is implied by the source reference location.)
2) source vector in LTF_S.

This method computes 2 output parameters:
1) coordinate of the common reference location in the target SRF, computed from the source reference location coordinate.
2) target vector in LTF_T.

It is invoked as follows:

```
target_SRF->transformVectorCommonOrigin (
        source_ref_coord,  /* input, Coord3D */
        source_vector,  /* input, Vector_3D */
        target_ref_coord,  /* output, Coord3D */
        target_vector);  /* output, Vector_3D */
```

7.3.3 Transform Vector in Body Frame

Given a vector in a body frame (or in general any linear reference frame, denoted by L), and given the orientation of this body frame with respect to a local tangent frame (LTF_S), the method `transformVectorInBodyFrame` computes the representation of the vector with respect to another local tangent frame (LTF_T), where LTF_S is the local tangent frame associated with the source SRF at the specified source reference location, and LTF_T is the local tangent frame associated with the target SRF at the specified target reference location. The output vector is computed by applying the composed orientation, from the orientation of LTF_S with respect to LTF_T with the source orientation, to the source vector. This method is equivalent to applying the orientation result from the `transformOrientation` method to the source vector. The invoking SRF is the target SRF.

This method takes 4 input parameters:

1) source reference location (a coordinate in the source SRF) where the origin of LTF_S is located. (In the C++ and Java implementations, the source SRF is implied by the source reference location.)
2) source orientation of some linear reference frame L with respect to LTF_S.
3) source vector in the linear reference frame L.
4) target reference location (a coordinate in this SRF, the target SRF) where the origin of the target local tangent frame (LTF_T) is located.

This method computes 1 output parameter:
1) target vector in LTF_T.

It is invoked as follows:

```
target_SRF->transformVectorInBodyFrame (
        source_ref_coord,  /* input, Coord3D */
        source_orientation,  /* input, Orientation */
        source_vector,  /* input, Vector_3D */
        target_ref_coord,  /* input, Coord3D */
        target_vector);  /* output, Vector_3D */
```

7.3.4 Transform Vector in Body Frame with Common Origin

Given a vector in a body frame (or in general any linear reference frame, denoted by L), and given the orientation of this body frame with respect to a local tangent frame (LTF_S), the method `transformVectorInBodyFrameCommonOrigin` computes the representation of the vector with respect to another local tangent frame (LTF_T), where LTF_S is the local tangent frame associated with the source SRF at the specified source reference location, and LTF_T is the local tangent frame associated with the target SRF at the specified target reference location. LTF_S and LTF_T have a common origin. The output vector is computed by applying the composed orientation, from the orientation of LTF_S with respect to LTF_T with the source orientation, to the source vector. This method is equivalent to applying the orientation result from the `transformOrientationCommonOrigin` method to the source vector. The invoking SRF is the target SRF.

This method takes 3 input parameters:
1) source reference location (a coordinate in the source SRF) where the origin of LTF_S is located. (In the C++ and Java implementations, the source SRF is implied by the source reference location.)
2) source orientation of some linear reference frame L with respect to LTF_S.
3) source vector in the linear reference frame L.

This method computes 2 output parameters:
1) coordinate of the common reference location in the target SRF, computed from the source reference location coordinate.
2) target vector in LTF_T.

It is invoked as follows:

```
target_SRF->transformVectorInBodyFrameCommonOrigin (
```

```
source_ref_coord,  /* input, Coord3D */
source_orientation,  /* input, Orientation */
source_vector,  /* input, Vector_3D */
target_ref_coord,  /* output, Coord3D */
target_vector);  /* output, Vector_3D */
```

7.4 Examples

The following examples show how to transform vector quantities between the various SRFs defined in Section 4.

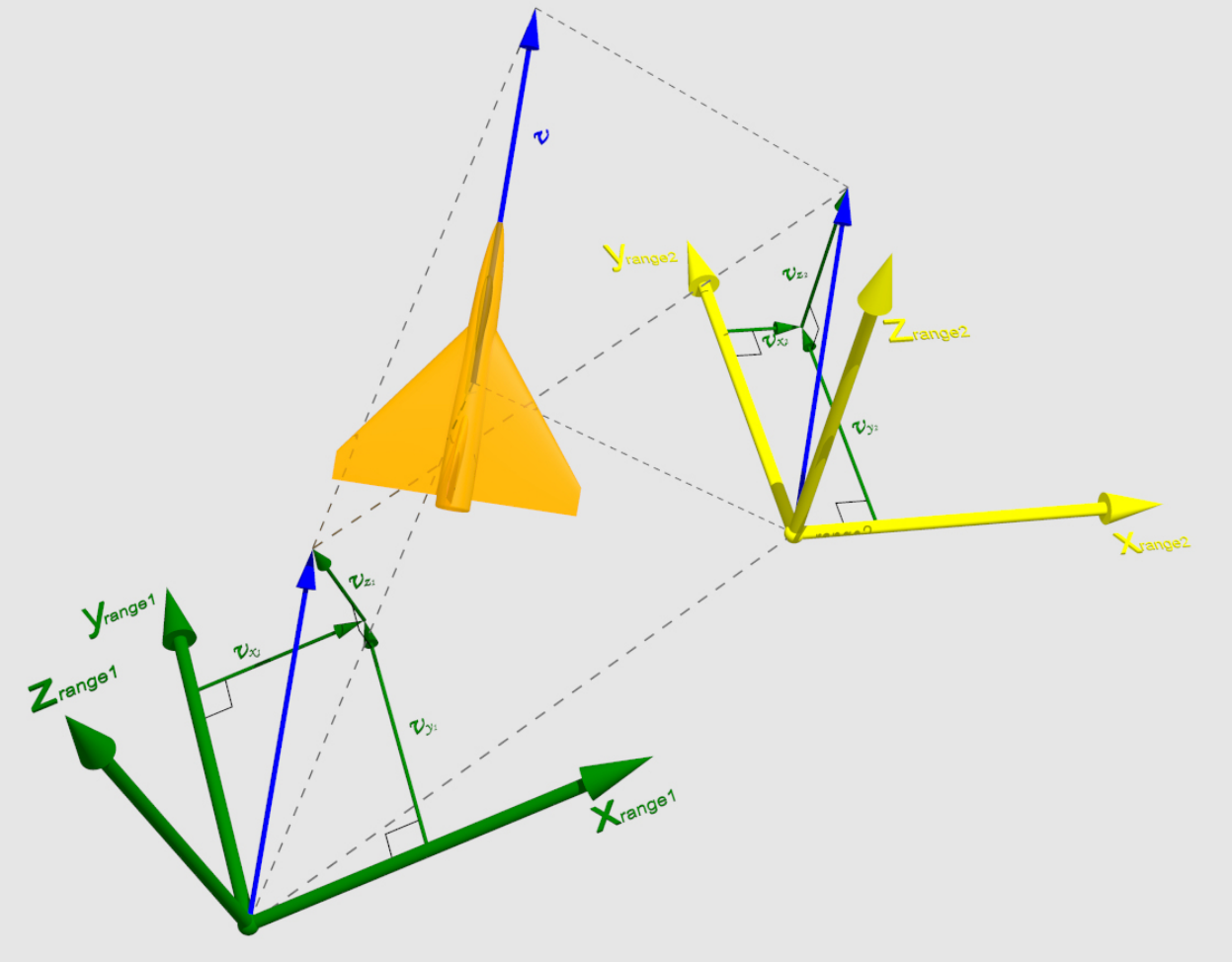

Figure 7-5. Vector Transformation from Range 1 to Range 2

7.4.1 Transform Between Range SRFs

Example 1: Given the linear velocity ***v***, with components (v_x, v_y, v_z), angular velocity ***ω***, with components (ωx, ωy, ωz), linear acceleration ***a*** (with components, (a_x, a_y, a_z), and angular acceleration ***α***, with components (α_x, α_y, α_z) of an aircraft, expressed with respect to the Range 1 SRF, transform these four vector quantities to the Range 2 SRF.

As shown in Figure 7-5, the corresponding axes of the two Range SRFs, shown in green and yellow, respectively, are not parallel to one another. Thus, the component values of a vector quantity ***v*** associated with the aircraft, representing, for example, its velocity with respect to the Range 1 SRF, are different from the corresponding component values of the same vector with

respect to the Range 2 SRF. The figure makes this clear by translating the vector ***v***, shown in blue, from the aircraft center of mass to the origins of each of the two Range SRFs, and showing the components of the vector, in green, with respect to each set of axes.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
3) Transform the vectors from the Range 1 SRF to the Range 2 SRF using the `transformVector` method of the Range 2 SRF object. This requires a reference coordinate for each of the two SRFs. Because the Range SRFs are both linear, any convenient locations can be chosen as the reference coordinates. Suppose the origins of the Range 1 and Range 2 SRFs, respectively, are chosen.

```
// From Range 1 to Range 2
Coord3D* range1_ref_coord = Range1_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

Coord3D* range2_ref_coord = Range2_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

// Aircraft linear velocity vector with respect to Range 1
// (200 m/sec ahead):
SRM_Vector_3D aircraft_range1_velocity;

aircraft_range1_velocity.array[0] = 200.0 * -0.4330127; // m/sec
aircraft_range1_velocity.array[1] = 200.0 * 0.75; // m/sec
aircraft_range1_velocity.array[2] = 200.0 * 0.5; // m/sec

SRM_Vector_3D aircraft_range2_velocity;

Range2_SRF->transformVector(
    *range1_ref_coord,
    aircraft_range1_velocity,
    *range2_ref_coord,
    aircraft_range2_velocity);

// Aircraft angular velocity vector with respect to Range 1
// (1 degree/sec clockwise roll):
SRM_Vector_3D aircraft_range1_angular_velocity;

aircraft_range1_angular_velocity.array[0] =
    1.0 * degreesToRadians * -0.4330127; // radians/sec
aircraft_range1_angular_velocity.array[1] =
    1.0 * degreesToRadians * 0.75; // radians/sec
aircraft_range1_angular_velocity.array[2] =
    1.0 * degreesToRadians * 0.5; // radians/sec

SRM_Vector_3D aircraft_range2_angular_velocity;

Range2_SRF->transformVector(
    *range1_ref_coord,
    aircraft_range1_angular_velocity,
    *range2_ref_coord,
    aircraft_range2_angular_velocity);
```

```
// Aircraft linear acceleration vector with respect to Range 1
// (10 m/sec**2 ahead):
SRM_Vector_3D aircraft_range1_acceleration;

aircraft_range1_acceleration.array[0] = 10.0 * -0.4330127; // m/sec^2
aircraft_range1_acceleration.array[1] = 10.0 * 0.75; // m/sec^2
aircraft_range1_acceleration.array[2] = 10.0 * 0.5; // m/sec^2

SRM_Vector_3D aircraft_range2_acceleration;

Range2_SRF->transformVector(
    *range1_ref_coord,
    aircraft_range1_acceleration,
    *range2_ref_coord,
    aircraft_range2_acceleration);

// Aircraft angular acceleration vector with respect to Range 1
// (1 degree/sec**2 clockwise roll):
SRM_Vector_3D aircraft_range1_angular_acceleration;

aircraft_range1_angular_acceleration.array[0] =
    1.0 * degreesToRadians * -0.4330127; // radians/sec^2
aircraft_range1_angular_acceleration.array[1] =
    1.0 * degreesToRadians * 0.75; // radians/sec^2
aircraft_range1_angular_acceleration.array[2] =
    1.0 * degreesToRadians * 0.5; // radians/sec^2

SRM_Vector_3D aircraft_range2_angular_acceleration;

Range2_SRF->transformVector(
    *range1_ref_coord,
    aircraft_range1_angular_acceleration,
    *range2_ref_coord,
    aircraft_range2_angular_acceleration);
```

7.4.2 Transform From Range to Geocentric

Example 2: Transform the linear velocity, linear acceleration, angular velocity, and angular acceleration vectors of a tank expressed with respect to the Range 1 SRF to the Geocentric WGS 1984 SRF.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Geocentric WGS 1984 SRF (see Section 4.1.2).
3) Transform the vectors from the Range 1 SRF to the Geocentric WGS 1984 SRF using the `transformVector` method of the Geocentric WGS 1984 SRF object. This requires a reference coordinate for each of the two SRFs. Because the Range 1 SRF and the Geocentric WGS 1984 SRF are both linear, any convenient locations can be chosen as the reference coordinates. Suppose the origin of the Range 1 SRF, and the origin of the Geocentric WGS 1984 SRF, are chosen.

```
// From Range 1 to Geocentric
Coord3D* range1_ref_coord = Range1_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

Coord3D* geocentric_ref_coord = Geocentric_WGS84_SRF->createCoordinate3D(
```

```
    0.0, 0.0, 0.0);

// Tank linear velocity vector with respect to Range 1
// (20 m/sec ahead):
SRM_Vector_3D tank_range1_velocity;

tank_range1_velocity.array[0] = 20.0 * 0.25; // m/sec
tank_range1_velocity.array[1] = 20.0 * 0.9330127; // m/sec
tank_range1_velocity.array[2] = 20.0 * 0.25881905; // m/sec

SRM_Vector_3D tank_geocentric_velocity;

Geocentric_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_velocity,
    *geocentric_ref_coord,
    tank_geocentric_velocity);

// Tank angular velocity vector with respect to Range 1
// (1 degree/sec right turn):
SRM_Vector_3D tank_range1_angular_velocity;

tank_range1_angular_velocity.array[0] =
    1.0 * degreesToRadians * 0.25; // radians/sec
tank_range1_angular_velocity.array[1] =
    1.0 * degreesToRadians * 0.9330127; // radians/sec
tank_range1_angular_velocity.array[2] =
    1.0 * degreesToRadians * 0.25881905; // radians/sec

SRM_Vector_3D tank_geocentric_angular_velocity;

Geocentric_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_angular_velocity,
    *geocentric_ref_coord,
    tank_geocentric_angular_velocity);

// Tank linear acceleration vector with respect to Range 1
// (-0.2 m/sec**2 ahead):
SRM_Vector_3D tank_range1_acceleration;

tank_range1_acceleration.array[0] = -0.2 * 0.25; // m/sec^2
tank_range1_acceleration.array[1] = -0.2 * 0.9330127; // m/sec^2
tank_range1_acceleration.array[2] = -0.2 * 0.25881905; // m/sec^2

SRM_Vector_3D tank_geocentric_acceleration;

Geocentric_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_acceleration,
    *geocentric_ref_coord,
    tank_geocentric_acceleration);

// Tank angular acceleration vector with respect to Range 1
// (-0.1 degree/sec right turn):
SRM_Vector_3D tank_range1_angular_acceleration;
```

```
    tank_range1_angular_acceleration.array[0] =
        -0.1 * degreesToRadians * 0.25; // radians/sec^2
    tank_range1_angular_acceleration.array[1] =
        -0.1 * degreesToRadians * 0.9330127; // radians/sec^2
    tank_range1_angular_acceleration.array[2] =
        -0.1 * degreesToRadians * 0.25881905; // radians/sec^2

    SRM_Vector_3D tank_geocentric_angular_acceleration;

    Geocentric_WGS84_SRF->transformVector(
        *range1_ref_coord,
        tank_range1_angular_acceleration,
        *geocentric_ref_coord,
        tank_geocentric_angular_acceleration);
```

7.4.3 Transform From Range to Geodetic

Example 3: Transform the linear velocity, linear acceleration, angular velocity, and angular acceleration vectors of a tank expressed with respect to the Range 1 SRF to the Geodetic WGS 1984 SRF.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
3) Transform the vectors from the Range 1 SRF to the Geodetic WGS 1984 SRF using the `transformVector` method of the Geodetic WGS 1984 SRF object. This requires a reference coordinate for each of the two SRFs. In this case, because the Range 1 SRF is a linear SRF, any convenient location can be chosen as the Range 1 reference coordinate. However, because the Geodetic WGS 1984 SRF is a curvilinear SRF, an appropriate and relevant location should be chosen as the geodetic reference coordinate. Suppose the origin of the Range 1 SRF is chosen as the reference coordinate for the Range 1 SRF, and is then converted to the Geodetic WGS 1984 SRF.

```
    // From Range 1 to Geodetic
    Coord3D* range1_ref_coord = Range1_SRF->createCoordinate3D(
        0.0, 0.0, 0.0);

    Coord3D* geodetic_ref_coord = Geodetic_WGS84_SRF->createCoordinate3D(
        0.0, 0.0, 0.0);

    SRM_Coordinate_Valid_Region region = Geodetic_WGS84_SRF-
>changeCoordinate3DSRF(
        *range1_ref_coord,
        *geodetic_ref_coord);

    // Tank linear velocity vector with respect to Range 1
    // (15 m/sec ahead):
    SRM_Vector_3D tank_range1_velocity;

    tank_range1_velocity.array[0] = 15.0 * 0.25; // m/sec
    tank_range1_velocity.array[1] = 15.0 * 0.9330127; // m/sec
    tank_range1_velocity.array[2] = 15.0 * 0.25881905; // m/sec

    SRM_Vector_3D tank_geodetic_velocity;
```

```
Geodetic_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_velocity,
    *geodetic_ref_coord,
    tank_geodetic_velocity);

// Tank angular velocity vector with respect to Range 1
// (1 degree/sec left turn):
SRM_Vector_3D tank_range1_angular_velocity;

tank_range1_angular_velocity.array[0] =
    -1.0 * degreesToRadians * 0.25; // radians/sec
tank_range1_angular_velocity.array[1] =
    -1.0 * degreesToRadians * 0.9330127; // radians/sec
tank_range1_angular_velocity.array[2] =
    -1.0 * degreesToRadians * 0.25881905; // radians/sec

SRM_Vector_3D tank_geodetic_angular_velocity;

Geodetic_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_angular_velocity,
    *geodetic_ref_coord,
    tank_geodetic_angular_velocity);

// Tank linear acceleration vector with respect to Range 1
// (0.1 m/sec**2 ahead):
SRM_Vector_3D tank_range1_acceleration;

tank_range1_acceleration.array[0] = 0.1 * 0.25; // m/sec^2
tank_range1_acceleration.array[1] = 0.1 * 0.9330127; // m/sec^2
tank_range1_acceleration.array[2] = 0.1 * 0.25881905; // m/sec^2

SRM_Vector_3D tank_geodetic_acceleration;

Geodetic_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_acceleration,
    *geodetic_ref_coord,
    tank_geodetic_acceleration);

// Tank angular acceleration vector with respect to Range 1
// (0.1 degree/sec left turn):
SRM_Vector_3D tank_range1_angular_acceleration;

tank_range1_angular_acceleration.array[0] =
    -0.1 * degreesToRadians * 0.25; // radians/sec^2
tank_range1_angular_acceleration.array[1] =
    -0.1 * degreesToRadians * 0.9330127; // radians/sec^2
tank_range1_angular_acceleration.array[2] =
    -0.1 * degreesToRadians * 0.25881905; // radians/sec^2

SRM_Vector_3D tank_geodetic_angular_acceleration;

Geodetic_WGS84_SRF->transformVector(
    *range1_ref_coord,
    tank_range1_angular_acceleration,
```

```
    *geodetic_ref_coord,
    tank_geodetic_angular_acceleration);
```

7.4.4 Transform From Geodetic to Geocentric

Example 4: Transform the linear velocity, linear acceleration, angular velocity, and angular acceleration vectors of an aircraft expressed with respect to the Geodetic WGS 1984 SRF to the Geocentric WGS 1984 SRF.

1) Create the SRF object for the Geodetic WGS 1984 SRF (see Section 4.1.1).
2) Create the SRF object for the Geocentric WGS 1984 SRF (see Section 4.1.2).
3) Transform the vectors from the Geodetic WGS 1984 SRF to the Geocentric WGS 1984 SRF using the `transformVector` method of the Geocentric WGS 1984 SRF object. Note that the entity state vectors for the Geodetic WGS 1984 SRF are specified in terms of the local tangent framedefined by the reference coordinate at the current position of the aircraft. This requires a reference coordinate for each of the two SRFs. In this case, because the Geodetic WGS 1984 SRF is a curvilinear SRF, an appropriate and relevant location should be chosen as the geodetic reference coordinate. Suppose the Geodetic WGS 1984 coordinate of the aircraft is chosen. This defines a local tangent frame with its origin located at the current position of the aircraft. Because the Geocentric WGS 1984 SRF is a linear SRF, any convenient location can be chosen as the geocentric reference coordinate. Suppose the geocentric origin is chosen as the geocentric reference coordinate.

```
// From Geodetic to Geocentric
SRM_Long_Float aircraft_longitude = -120.5 * degreesToRadians,
    aircraft_latitude = 33.5 * degreesToRadians,
    aircraft_ellipsoidal_height = 5000.0; // meters

Coord3D* geodetic_ref_coord = Geodetic_WGS84_SRF->createCoordinate3D(
    aircraft_longitude,
    aircraft_latitude,
    aircraft_ellipsoidal_height);

Coord3D* geocentric_ref_coord = Geocentric_WGS84_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

// Aircraft linear velocity vector with respect to Geodetic WGS84 SRF
// (200 m/sec northeast):
SRM_Vector_3D aircraft_geodetic_velocity;

aircraft_geodetic_velocity.array[0] = 200.0 * 0.70710678; // m/sec
aircraft_geodetic_velocity.array[1] = 200.0 * 0.70710678; // m/sec
aircraft_geodetic_velocity.array[2] = 200.0 * 0.0; // m/sec

SRM_Vector_3D aircraft_geocentric_velocity;

Geocentric_WGS84_SRF->transformVector(
    *geodetic_ref_coord,
    aircraft_geodetic_velocity,
    *geocentric_ref_coord,
    aircraft_geocentric_velocity);

// Aircraft angular velocity vector with respect to Geodetic WGS84 SRF
```

```
// (1 degree/sec right turn):
SRM_Vector_3D aircraft_geodetic_angular_velocity;

aircraft_geodetic_angular_velocity.array[0] =
    -1.0 * degreesToRadians * 0.0; // radians/sec
aircraft_geodetic_angular_velocity.array[1] =
    -1.0 * degreesToRadians * 0.0; // radians/sec
aircraft_geodetic_angular_velocity.array[2] =
    -1.0 * degreesToRadians * 1.0; // radians/sec

SRM_Vector_3D aircraft_geocentric_angular_velocity;

Geocentric_WGS84_SRF->transformVector(
    *geodetic_ref_coord,
    aircraft_geodetic_angular_velocity,
    *geocentric_ref_coord,
    aircraft_geocentric_angular_velocity);

// Aircraft linear acceleration vector with respect to Geodetic WGS84 SRF
// (10 m/sec**2 ahead):
SRM_Vector_3D aircraft_geodetic_acceleration;

aircraft_geodetic_acceleration.array[0] = 10.0 * 0.70710678; // m/sec^2
aircraft_geodetic_acceleration.array[1] = 10.0 * 0.70710678; // m/sec^2
aircraft_geodetic_acceleration.array[2] = 10.0 * 0.0; // m/sec^2

SRM_Vector_3D aircraft_geocentric_acceleration;

Geocentric_WGS84_SRF->transformVector(
    *geodetic_ref_coord,
    aircraft_geodetic_acceleration,
    *geocentric_ref_coord,
    aircraft_geocentric_acceleration);

// Aircraft angular acceleration vector with respect to Geodetic WGS84
//  SRF (1 degree/sec**2 right turn):
SRM_Vector_3D aircraft_geodetic_angular_acceleration;

aircraft_geodetic_angular_acceleration.array[0] =
    -1.0 * degreesToRadians * 0.0; // radians/sec^2
aircraft_geodetic_angular_acceleration.array[1] =
    -1.0 * degreesToRadians * 0.0; // radians/sec^2
aircraft_geodetic_angular_acceleration.array[2] =
    -1.0 * degreesToRadians * 1.0; // radians/sec^2

SRM_Vector_3D aircraft_geocentric_angular_acceleration;

Geocentric_WGS84_SRF->transformVector(
    *geodetic_ref_coord,
    aircraft_geodetic_angular_acceleration,
    *geocentric_ref_coord,
    aircraft_geocentric_angular_acceleration);
```

7.4.5 Transform From Aircraft Body Frame to Range 1

Example 5: Transform the linear velocity, linear acceleration, angular velocity, and angular acceleration vectors of an aircraft from the aircraft's body frame to the Range 1 SRF. It is

assumed that, at any given instant in time, the orientation of the aircraft body frame with respect to the Range 1 SRF, $\mathbf{\Omega}_{BodyToRange1}$, can be computed to sufficient accuracy using the information provided by the inertial systems on the aircraft. The inertial systems provide aircraft orientation information in the form of a set of Tait-Bryan angles, i.e., roll, pitch, and yaw. It is further assumed that the aircraft inertial systems have been calibrated to the Range 1 origin local North and local "up" directions.

This example uses an alternative to the transformation procedure given in Section 7.3. This alternative procedure does not require the source SRF, which is the aircraft's body frame, to be explicitly defined.

1) Create the orientation object for the aircraft body frame with respect to test Range 1 SRF (see Section 6.2).

Given the assumptions above, this can be accomplished in three stages. First, an orientation object is created that represents the orientation of the aircraft body frame with respect to the "calibration frame" of its inertial systems. This is accomplished using the Tait-Bryan roll, pitch, and yaw angles reported by the inertial systems:

```
SRM_Tait_Bryan_Angles_Params aircraft_tait_bryan_params;

aircraft_tait_bryan_params.roll = 5.0 * degreesToRadians;
aircraft_tait_bryan_params.pitch = -5.0 * degreesToRadians;
aircraft_tait_bryan_params.yaw = 90.0 * degreesToRadians;

OrientationTaitBryanAngles orientation_body_to_calibration(
    aircraft_tait_bryan_params);
```

In the calibration frame, the aircraft *x*-axis points toward local North and the *z*-axis points toward local down, while the test Range 1 *x*-axis points toward local East and the *z*-axis points toward local up. A second orientation object that represents the orientation of the calibration frame with respect to test Range 1 can therefore be created as follows:

```
// From Aircraft Body Frame to Range 1
SRM_Matrix_3x3 matrix_params_calibration_to_range1 = {
    0.0, 1.0, 0.0,
    1.0, 0.0, 0.0,
    0.0, 0.0, -1.0};

OrientationMatrix orientation_calibration_to_range1(
    matrix_params_calibration_to_range1);
```

These two orientation objects, representing orientation of the aircraft body frame with respect to the calibration frame, and the orientation of the calibration frame with respect to the Range 1 SRF, can now be composed to form the orientation of the aircraft body frame with respect to the Range 1 SRF:

```
OrientationMatrix orientation_body_to_range1 =
    OrientationMatrix::compose(orientation_calibration_to_range1,
        orientation_body_to_calibration);
```

2) Transform the vectors from the aircraft's body frame to the Range 1 SRF using the `transformVector` method of the orientation object resulting from step 2 above. This method transforms a vector quantity from the source SRF of the orientation object (i.e., the aircraft's body frame) to its target SRF (i.e., the Range 1 SRF). Note that because this orientation object captures the relationship between the aircraft body frame and the Range 1 SRF, the actual SRF objects are not needed to perform this transformation.

```
// Aircraft body velocity vector (100 m/sec ahead):
SRM_Vector_3D aircraft_body_velocity;

aircraft_body_velocity.array[0] = 100.0; // m/sec
aircraft_body_velocity.array[1] = 0.0; // m/sec
aircraft_body_velocity.array[2] = 0.0; // m/sec

SRM_Vector_3D aircraft_range1_velocity =
    orientation_body_to_range1.transformVector(
        aircraft_body_velocity);

// Aircraft body angular velocity vector
// (2 degrees/sec clockwise roll):
SRM_Vector_3D aircraft_body_angular_velocity;

aircraft_body_angular_velocity.array[0] =
    2.0 * degreesToRadians * 1.0; // radians/sec
aircraft_body_angular_velocity.array[1] =
    2.0 * degreesToRadians * 0.0; // radians/sec
aircraft_body_angular_velocity.array[2] =
    2.0 * degreesToRadians * 0.0; // radians/sec

SRM_Vector_3D aircraft_range1_angular_velocity =
    orientation_body_to_range1.transformVector(
        aircraft_body_angular_velocity);

// Aircraft body acceleration vector
// (5 m/sec ahead; 1 m/sec upward):
SRM_Vector_3D aircraft_body_acceleration;

aircraft_body_acceleration.array[0] = 5.0; // m/sec^2
aircraft_body_acceleration.array[1] = 0.0; // m/sec^2
aircraft_body_acceleration.array[2] = -1.0; // m/sec^2

SRM_Vector_3D aircraft_range1_acceleration =
    orientation_body_to_range1.transformVector(
        aircraft_body_acceleration);

// Aircraft body angular acceleration vector
// (+0.1 degrees/sec**2 clockwise roll):
SRM_Vector_3D aircraft_body_angular_acceleration;

aircraft_body_angular_acceleration.array[0] =
    0.1 * degreesToRadians * 1.0; // radians/sec^2
aircraft_body_angular_acceleration.array[1] =
    0.1 * degreesToRadians * 0.0; // radians/sec^2
aircraft_body_angular_acceleration.array[2] =
    0.1 * degreesToRadians * 0.0; // radians/sec^2
```

```
SRM_Vector_3D aircraft_range1_angular_acceleration =
    orientation_body_to_range1.transformVector(
        aircraft_body_angular_acceleration);
```

7.4.6 Transform From Aircraft Body Frame to Range 2

Example 6: Transform the linear velocity, linear acceleration, angular velocity, and angular acceleration vectors of the aircraft in the previous example from the aircraft's body frame to the Range 2 SRF. The inertial systems on the aircraft were calibrated with respect to local North and up at the Range 1 origin. These directions differ from local North and up at the Range 2 origin. The orientation $\boldsymbol{\Omega}_{BodyToRange1}$ computed in the previous example can be used in this case to directly transform vector quantities in the aircraft body frame to the Range 2 SRF using the `transformVectorInBodyFrame` method.

1) Create the SRF object for the Range 1 SRF (see Section 4.2.1).
2) Compute the orientation object for the aircraft body frame with respect to the Range 1 SRF (see example 5).
3) Create the SRF object for the Range 2 SRF (see Section 4.2.1).
4) Transform the vectors from the aircraft's body frame to the Range 2 SRF using the `transformVectorInBodyFrame` method of the Range 2 SRF object. This method uses the orientation of the aircraft body frame with respect to the Range 1 SRF as a parameter. This method also requires a reference coordinate for each of the two Range SRFs. Because the Range SRFs are both linear, any convenient locations can be chosen as the reference coordinates. Suppose the origins of the Range 1 and Range 2 SRFs, respectively, are chosen.

```
// From Aircraft Body Frame to Range 2
Coord3D* range1_ref_coord = Range1_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

Coord3D* range2_ref_coord = Range2_SRF->createCoordinate3D(
    0.0, 0.0, 0.0);

SRM_Tait_Bryan_Angles_Params aircraft_tait_bryan_params;

aircraft_tait_bryan_params.roll = 5.0 * degreesToRadians;
aircraft_tait_bryan_params.pitch = -5.0 * degreesToRadians;
aircraft_tait_bryan_params.yaw = 90.0 * degreesToRadians;

OrientationTaitBryanAngles orientation_body_to_calibration(
    aircraft_tait_bryan_params);

SRM_Matrix_3x3 matrix_params_calibration_to_range1 = {
    0.0, 1.0, 0.0,
    1.0, 0.0, 0.0,
    0.0, 0.0, -1.0};

OrientationMatrix orientation_calibration_to_range1(
    matrix_params_calibration_to_range1);

OrientationMatrix orientation_body_to_range1 =
    OrientationMatrix::compose(orientation_calibration_to_range1,
        orientation_body_to_calibration);
```

```
// Aircraft body vector quantities, with values as in Example 5

// Aircraft body velocity vector (100 m/sec ahead):
SRM_Vector_3D aircraft_body_velocity;

aircraft_body_velocity.array[0] = 100.0; // m/sec
aircraft_body_velocity.array[1] = 0.0; // m/sec
aircraft_body_velocity.array[2] = 0.0; // m/sec

// Aircraft body angular velocity vector
// (2 degrees/sec clockwise roll):
SRM_Vector_3D aircraft_body_angular_velocity;

aircraft_body_angular_velocity.array[0] =
    2.0 * degreesToRadians * 1.0; // radians/sec
aircraft_body_angular_velocity.array[1] =
    2.0 * degreesToRadians * 0.0; // radians/sec
aircraft_body_angular_velocity.array[2] =
    2.0 * degreesToRadians * 0.0; // radians/sec

// Aircraft body acceleration vector
// (5 m/sec ahead; 1 m/sec upward):
SRM_Vector_3D aircraft_body_acceleration;

aircraft_body_acceleration.array[0] = 5.0; // m/sec^2
aircraft_body_acceleration.array[1] = 0.0; // m/sec^2
aircraft_body_acceleration.array[2] = -1.0; // m/sec^2

// Aircraft body angular acceleration vector
// (+0.1 degrees/sec**2 clockwise roll):
SRM_Vector_3D aircraft_body_angular_acceleration;

aircraft_body_angular_acceleration.array[0] =
    0.1 * degreesToRadians * 1.0; // radians/sec^2
aircraft_body_angular_acceleration.array[1] =
    0.1 * degreesToRadians * 0.0; // radians/sec^2
aircraft_body_angular_acceleration.array[2] =
    0.1 * degreesToRadians * 0.0; // radians/sec^2

SRM_Vector_3D aircraft_range2_velocity;

Range2_SRF->transformVectorInBodyFrame(
    *range1_ref_coord,
    orientation_body_to_range1,
    aircraft_body_velocity,
    *range2_ref_coord,
    aircraft_range2_velocity);

SRM_Vector_3D aircraft_range2_angular_velocity;

Range2_SRF->transformVectorInBodyFrame(
    *range1_ref_coord,
    orientation_body_to_range1,
    aircraft_body_angular_velocity,
    *range2_ref_coord,
    aircraft_range2_angular_velocity);
```

```
SRM_Vector_3D aircraft_range2_acceleration;

Range2_SRF->transformVectorInBodyFrame(
    *range1_ref_coord,
    orientation_body_to_range1,
    aircraft_body_acceleration,
    *range2_ref_coord,
    aircraft_range2_acceleration);

SRM_Vector_3D aircraft_range2_angular_acceleration;

Range2_SRF->transformVectorInBodyFrame(
    *range1_ref_coord,
    orientation_body_to_range1,
    aircraft_body_angular_acceleration,
    *range2_ref_coord,
    aircraft_range2_angular_acceleration);
```

To transform vector quantities from the aircraft body frame to Geocentric, substitute Geocentric_WGS84_SRF for Range2_SRF in this example.

An alternate method to realize example 6 is to transform the orientation $\mathbf{\Omega}_{BodyToRange1}$ to be with respect to Range 2 and then use the methods of example 5. In particular, $\mathbf{\Omega}_{BodyToRange2}$ may be computed with:

```
Orientation* orientation_body_to_range2 = new OrientationMatrix ();
Range2_SRF->transformOrientation (
        *range1_ref_coord,
        *orientation_body_to_range1,
        *range2_ref_coord,
        *orientation_body_to_range2);
```