OGC™ GeoPose 1.0: C#

Table of Contents

Introduction

NOTE	Latest version 15 February 2023. This documentation is maintained by Steve Smyth steve@opensiteplan.org If you have questions, please send them to me and I will answer them and/or add the answer, correction, or clarification to tis document.

A frequent question is "How should I implement software that makes or manipulates OGC GeoPoses?" This tutorial is one answer. It may not describe exactly the software you may need to build but I hope it gives you the necessary building blocks to see how you might proceed.

This tutorial describes one template for developing a software implementation supporting GeoPose 1.0 using any one of many modern object-oriented programming languages. The focus here is on a C# implementation but the principles are very similar for most programming languages supporting object-oriented design.

OGC GeoPose 1.0 is a draft international standard for a family of JSON data objects representing the position and orientation of real or virtual entities. There are eight independent forms described in the standard. There are no dependencies between the forms. Each may be implemented independently. There is no requirement to implement any specific form or number of forms.

This tutorial only touches briefly on the **use** of GeoPose - another important topic.

Use Cases

Here are some examples of use cases where GeoPose can play a role.

The GeoPose use cases involve interactions between information systems or between an information system and a storage medium. The essential role of a GeoPose is to convey the position and orientation of a real or virtual object. The possibility of chained transformational relationships and cross-linkages between chains affords representation of complex pose relationships and a way to bring a collection of related GeoPoses in a common geographic reference frame.

Augmented and Mixed Reality

Augmented Reality (AR) integrates synthetic objects or synthetic representations of real objects with a physical environment. Geospatial AR experiences can use GeoPose to position synthetic objects or their representations in the physical environment. The geospatial connection provides a common reference frame to support integration in AR.

- Stored representation of synthetic objects
- Positioning information to support integration of synthetic object data in a representation or visualization of the physical environment
- Report of position and orientation from a mobile device to an AR network service
- Input to visual occlusion calculations
- Input to ray-casting and line-of-sight calculations
- Input to proximity calculations
- Input and output to and from trajectory projection calculations

Mobile Autonomy

Autonomous vehicles are mobile objects that move through water, across a water surface, in the air, through the solid earth (tunnel boring machine), on the land surface, or in outer space without real-time control by an independent onboard operator. A pose captures the essential information in positioning and orienting a moving object. Sensors attached to mobile elements have their own poses and a chain of reference frame transformations enables common reference frames to be used for data fusion. The possibility of relating the vehicle to other elements of the environment via a common reference frame is essential.

- Provide accurate visual positioning and guidance based on one or more services based on a 3D representation of the real world combined with real time detection and location of real world objects
- Calculate parameters such as distances and routes
- Record the trajectory of a moving vehicle.

Built Environment

The built environment consists of objects constructed by humans and located in physical space. Buildings, roads, dams, railways, and underground utilities are all part of the built environment. The location and orientation of built objects, especially those whose view is occluded by other objects is essential information needed for human interaction with the built environment. A common reference frame tied to the earth's surface facilitates the integration of these objects when their representations are supplied by different sources.

- Specify the position and orientation of visible objects and objects that are underground or hidden within a construction.
- Compactly and consistently specify or share the location and pose of objects in architecture, design and construction.

Simulations, Replicas, and the Metaverse

Synthetic environments contain collections of moving objects, which themselves may be composed of connected and articulated parts, in an animation or simulation environment that contains a fixed background of air, land, water, vegetation, built objects, and other non-moving elements. The assembly is animated over some time period to provide visualizations or analytical results of the evolving state of the modelled environment. Synthetic environments support training, rehearsal, and archival of activities and events. The location and orientation of the movable elements of a scene are the key data controlling animation of in a synthetic environment. Since there may be multiple possible animations consistent with observations, storage of the sequences of poses of the actors, vehicles, and other objects is a direct and compact way of representing the variable aspects of the event. Access to one or more common reference frames through a graph of frame transformations makes a coherent assembly possible.

- Record pose relationships of all mobile elements in an environment
- Track targets from a moving platform

- Control animation of mobile elements in an environment using stored pose time sequences

Image Understanding

3D image understanding is the segmentation of an image or sequence of images into inferred 3D objects in specific semantic categories, possibly determining or constraining their motion and/or geometry. One important application of image understanding is the recognition of moving elements in a time series of images. A pose is a compact representation of the key geometric characteristics of a moving element. In addition to moving elements sensed by an imaging device, it is often useful to know the pose of the sensor or imaging device itself. A common geographic reference frame integrates the objects into a single environment.

- Instantaneous and time series locations and orientations of mobile objects
- Instantaneous and time series location and orientation of an optical and/or depth imaging device using Simultaneous Location And Mapping (SLAM)
- Instantaneous and time series estimation of the changes in location and orientation of an object using an optical imaging device (Visual Odometry)
- Instantaneous and time series location and orientation of an optical imaging device used for photogrammetry

Goals of this Tutorial

The OGC GeoPose 1.0 standard does not specify anything about software design or programming language. The primary goal of this tutorial is to walk through a design and implementation of software that works well with OGC GeoPose 1.0 and which can be integrated in to applications that create or receive GeoPose 1.0 data objects. The only requirement is that the language offer basic object-oriented programming support.

NOTE	There is also a TypeScript version of this tutorial.

There are several specific goals:

Add GeoPose to applications

An example library makes it less difficult to start quickly and have a level of confidence that the operations are performed correctly. The answers to many practical questions can be found in the code.

Linkage to WKT parsers and coordinate transformation libraries

GeoPose is based on an abstraction of transformations linking pairs of spaces or their associated reference frames. Many of the definitions of reference frames are complex and described in terms specific to a particular discipline, such as geodesy, surveying, or astrophysics. Experts in these disciplines have built specialized databases and transformation software. It is highly desirable to be

able to use their work.

One very useful example is the PROJ coordinate transformation library either used by itself or as part of the Geospatial Data Abstraction Library (GDAL) library. This tutorial uses an interface to PROJ to implement a range of more general transformations.

Many frame specifications follow ISO 19111 and can be expressed as "well-known-text" structures that define datum, coordinate system, and transformation methods. Linkage to mature libraries such as GDAL and PROJ can also eliminate the need to parse and interpret these specialized structures within a GeoPose implementation.

Experiment with possible new features

Having a working implementation of the standardized elements of GeoPose 1.0 makes it easy to experiment wih new features that might be proposed for a new version of the standard. I give two examples of how this can be done. First, I have provided three new properties for the Basic and Advanced GeoPoses that have proved to be useful in my GeoPose applications. These additional properties serialize as additional JSON properties, which are explicitly allowed by the standard. Second, I have included the "Local" (Geo)Pose. Local is the closest to the usual concept of a pose in computer graphics. It is designed to allow chains and trees in the space of the rotated local tangent plane, east-north-up Cartesian coordinate system associated with the inner frame of Basic GeoPoses. The Local GeoPose can be expressed as an Advanced GeoPose but creating a simplified version with the frame transformation hardwired makes for clearer programming. I have not done so in this tutorial but it would be possible to configure the JSON serialization to output the Advanced equivalent, rather than a non-standard form.

Template applies to a wide range of object-oriented languages

The design only relies on a few basic O-O concepts and capabilities. These are supported by a wide range of old and new languages. In this and a companion C# post, I will cover **TypeScript 4.9.5** and **C# 11 - .NET 6**. In future posts, I will continue with some or all of C++, Java, Swift, Kotlin, and Python.

Architecture

Goals

There are many possible implementations. My primary consideration is a simple and completely hierarchical design - patterned to meet the capabilities of common object-oriented languages. I also wanted to make it possible to consider individual parts in isolation and then to assemble them into a GeoPose inheritance tree.

I describe the parts in reverse order of dependency. By the time you get to the Abstract GeoPose, there will be enough elements to start assembling them into the final structures.

Design Patterns

There are two obvious patterns to follow in a GeoPose implementation:

1. implement the Structural Data Units, which abstract the individual data objects defined in the GeoPose 1.0 standard - build an "Adapter" - supporting serialization and deserialization of GeoPose data objects or
2. implement the core structures of an anchored pose, focusing on implementation of the frame transforms and rotational transforms - build an "Engine" - and including conforming serialization and/or deserialization.

The Adapter pattern is straightforward to implement and does not require a deep understanding of the details of specification and implementation of the transformations. The pattern that I use here is the Engine, where the intent is to support a range of useful frame and rotational transformations.

Eight Steps

The development steps outlined here proceed from independent components to three categories of GeoPoses: Basic, Advanced, and Local. Note that Local GeoPoses are within the scope of the GeoPose 1.0 logical model but must be serialized as Advanced GeoPoses to be compliant data objects.

- Step 1: Supporting Datatypes and Functions
- Step 2: Positions
- Step 3: Frame Transforms
- Step 4: Orientations
- Step 5: Abstract GeoPose
- Step 6: Basic GeoPoses
- Step 7: Advanced GeoPose
- Step 8: Local Pose

C# implementation

The following is the sequence of steps for a C# implementation:

Step 1: Supporting Datatypes and Functions

Start here.

There are two simple datatypes that encapsulate an identifier and a time instant: PoseID and TimeValue. They are used in several of the classes. They are separated out because their design is dependent on the application domain and the need to interoperate with other systems. The GeoPose 1.0 standard does not specify any identifier and it defines a "valid Time" for only some of the GeoPose forms. Experience with the GeoPose since the initial publication shows the utility of references to GeoPoses and to having times associated with many individual GeoPoses.

Note that additional (private) properties may be added to most otherwise compliant GeoPose elements.

PoseID

PoseID has a single property - an id string.

UnixTime

UnixTime has a single property - a string representation of the number of Unix time seconds multiplied by 1 000 for millisecond resolution.

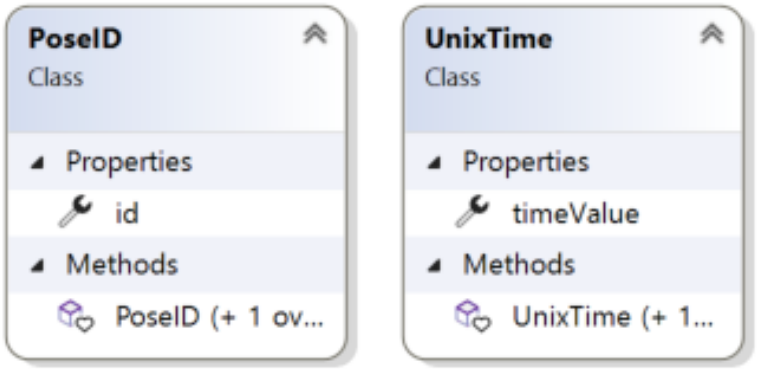

Figure 1. The PoseID and UnixTime Extras Classes

Coordinate conversion

The methods of the LTP_ENU class are needed to support the Basic and Advanced classes' frame transformations. The GeoPose implementations must implement the actual transformations implied or designated by the class or outer and inner frame definitions. This in contrast to the GeoPose data objects, which carry no explicit information about how the transformations should be carried out.

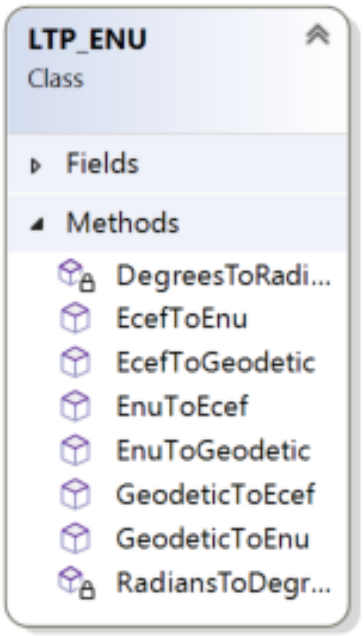

Figure 2. Calculation Support Classes

The calculation support classes are not needed to create or consume GeoPose data objects but they **are** needed to actually use the GeoPoses in an application.

C# implementation:

Datatypes

Copyright and License (MIT)

```
// Implementation step: 1 - start here.
// These classes are non-structural elements.
// These are part of optional elements that are allowed but not standardized.
using System;

namespace Extras
{
    public class PoseID
    {
        internal PoseID()
        {

        }
        public PoseID(string id)
        {
            this.id = id;
        }
        public string id { get; set; } = string.Empty;
    }
    public class UnixTime
    {
        internal UnixTime()
        {

        }
        // Constructor from long integer count of UNIX Time seconds x 1000
        public UnixTime(long longTime)
        {
            timeValue = longTime.ToString();
        }
        public string timeValue { get; set; } = string.Empty;
    }
}
```

LTP_ENU coordinate conversion

Copyright and License (MIT)

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

using ProjNet;
using ProjNet.CoordinateSystems;
using ProjNet.CoordinateSystems.Transformations;
using GeoAPI.CoordinateSystems.Transformations;
```

```
namespace Support
{
    // This is an implementation of EPSG method 9837 using sections 4.1.1 and 4.1.2 of
https://www.iogp.org/wp-content/uploads/2019/09/373-07-02.pdf.
    public class LTP_ENU
    {
        // WGS-84 geodetic constants
        const double a = 6378137.0;         // WGS-84 Earth semimajor axis (m)

        const double b = 6356752.314245;     // Derived Earth semiminor axis (m)
        const double f = (a - b) / a;           // Ellipsoid Flatness
        const double f_inv = 1.0 / f;       // Inverse flattening

        //const double f_inv = 298.257223563; // WGS-84 Flattening Factor of the Earth
        //const double b = a - a / f_inv;
        //const double f = 1.0 / f_inv;

        const double a_sq = a * a;
        const double b_sq = b * b;
        const double e_sq = f * (2 - f);    // Square of Eccentricity

        // Converts WGS-84 Geodetic point (lat, lon, h) to the
        // Earth-Centered Earth-Fixed (ECEF) coordinates (x, y, z).
        public static void GeodeticToEcef(double lat, double lon, double h,
                                          out double x, out double y, out double z)
        {
            // Convert to radians in notation consistent with the paper:
            var lambda = DegreesToRadians(lat);
            var phi = DegreesToRadians(lon);
            var s = Math.Sin(lambda);
            var N = a / Math.Sqrt(1 - e_sq * s * s);

            var sin_lambda = Math.Sin(lambda);
            var cos_lambda = Math.Cos(lambda);
            var cos_phi = Math.Cos(phi);
            var sin_phi = Math.Sin(phi);

            x = (h + N) * cos_lambda * cos_phi;
            y = (h + N) * cos_lambda * sin_phi;
            z = (h + (1 - e_sq) * N) * sin_lambda;
        }

        // Converts the Earth-Centered Earth-Fixed (ECEF) coordinates (x, y, z) to
        // (WGS-84) Geodetic point (lat, lon, h).
        public static void EcefToGeodetic(double x, double y, double z,
                                          out double lat, out double lon, out double
h)
        {
            var eps = e_sq / (1.0 - e_sq);
            var p = Math.Sqrt(x * x + y * y);
            var q = Math.Atan2((z * a), (p * b));
```

```
            var sin_q = Math.Sin(q);
            var cos_q = Math.Cos(q);
            var sin_q_3 = sin_q * sin_q * sin_q;
            var cos_q_3 = cos_q * cos_q * cos_q;
            var phi = Math.Atan2((z + eps * b * sin_q_3), (p - e_sq * a * cos_q_3));
            var lambda = Math.Atan2(y, x);
            var v = a / Math.Sqrt(1.0 - e_sq * Math.Sin(phi) * Math.Sin(phi));
            h = (p / Math.Cos(phi)) - v;

            lat = RadiansToDegrees(phi);
            lon = RadiansToDegrees(lambda);
        }

        // Converts the Earth-Centered Earth-Fixed (ECEF) coordinates (x, y, z) to
        // East-North-Up coordinates in a Local Tangent Plane that is centered at the
        // (WGS-84) Geodetic point (lat0, lon0, h0).
        public static void EcefToEnu(double x, double y, double z,
                                     double lat0, double lon0, double h0,
                                     out double xEast, out double yNorth, out
double zUp)
        {
            // Convert to radians in notation consistent with the paper:
            var lambda = DegreesToRadians(lat0);
            var phi = DegreesToRadians(lon0);
            var s = Math.Sin(lambda);
            var N = a / Math.Sqrt(1 - e_sq * s * s);

            var sin_lambda = Math.Sin(lambda);
            var cos_lambda = Math.Cos(lambda);
            var cos_phi = Math.Cos(phi);
            var sin_phi = Math.Sin(phi);

            double x0 = (h0 + N) * cos_lambda * cos_phi;
            double y0 = (h0 + N) * cos_lambda * sin_phi;
            double z0 = (h0 + (1 - e_sq) * N) * sin_lambda;

            double xd, yd, zd;
            xd = x - x0;
            yd = y - y0;
            zd = z - z0;

            // This is the matrix multiplication
            xEast = -sin_phi * xd + cos_phi * yd;
            yNorth = -cos_phi * sin_lambda * xd - sin_lambda * sin_phi * yd +
cos_lambda * zd;
            zUp = cos_lambda * cos_phi * xd + cos_lambda * sin_phi * yd + sin_lambda *
zd;
        }

        // Inverse of EcefToEnu. Converts East-North-Up coordinates (xEast, yNorth,
zUp) in a
```

```
        // Local Tangent Plane that is centered at the (WGS-84) Geodetic point (lat0,
lon0, h0)
        // to the Earth-Centered Earth-Fixed (ECEF) coordinates (x, y, z).
        public static void EnuToEcef(double xEast, double yNorth, double zUp,
                                     double lat0, double lon0, double h0,
                                     out double x, out double y, out double z)
        {
            // Convert to radians in notation consistent with the paper:
            var lambda = DegreesToRadians(lat0);
            var phi = DegreesToRadians(lon0);
            var s = Math.Sin(lambda);
            var N = a / Math.Sqrt(1 - e_sq * s * s);

            var sin_lambda = Math.Sin(lambda);
            var cos_lambda = Math.Cos(lambda);
            var cos_phi = Math.Cos(phi);
            var sin_phi = Math.Sin(phi);

            double x0 = (h0 + N) * cos_lambda * cos_phi;
            double y0 = (h0 + N) * cos_lambda * sin_phi;
            double z0 = (h0 + (1 - e_sq) * N) * sin_lambda;

            double xd = -sin_phi * xEast - cos_phi * sin_lambda * yNorth + cos_lambda
* cos_phi * zUp;
            double yd = cos_phi * xEast - sin_lambda * sin_phi * yNorth + cos_lambda *
sin_phi * zUp;
            double zd = cos_lambda * yNorth + sin_lambda * zUp;

            x = xd + x0;
            y = yd + y0;
            z = zd + z0;
        }

        // Converts the geodetic WGS-84 coordinated (lat, lon, h) to
        // East-North-Up coordinates in a Local Tangent Plane that is centered at the
        // (WGS-84) Geodetic point (lat0, lon0, h0).
        public static void GeodeticToEnu(double lat, double lon, double h,
                                         double lat0, double lon0, double h0,
                                         out double xEast, out double yNorth, out
double zUp)
        {
            double x, y, z;
            GeodeticToEcef(lat, lon, h, out x, out y, out z);
            EcefToEnu(x, y, z, lat0, lon0, h0, out xEast, out yNorth, out zUp);
        }
        // Converts the geodetic WGS-84 coordinated (lat, lon, h) to
        // East-North-Up coordinates in a Local Tangent Plane that is centered at the
        // (WGS-84) Geodetic point (lat0, lon0, h0).
        public static Positions.CartesianPosition
GeodeticToEnu(Positions.GeodeticPosition geodeticPosition,
            Positions.GeodeticPosition tangentPosition)
```

```
        {
            double x, y, z;
            double xEast, yNorth, zUp;
            GeodeticToEcef(geodeticPosition.lat, geodeticPosition.lon,
geodeticPosition.h, out x, out y, out z);
            EcefToEnu(x, y, z, tangentPosition.lat, tangentPosition.lon,
tangentPosition.h, out xEast, out yNorth, out zUp);
            Positions.CartesianPosition enuPosition = new
Positions.CartesianPosition(xEast, yNorth, zUp);
            return enuPosition;
        }
        public static void EnuToGeodetic(double xEast, double yNorth, double zUp,
                                          double lat0, double lon0, double h0,
                                         out double lat, out double lon, out double
h
                                         )
        {
            double x, y, z;
            EnuToEcef(xEast, yNorth, zUp, lat0, lon0, h0, out x, out y, out z);
            EcefToGeodetic(x, y, z, out lat, out lon, out h);
        }

        static double DegreesToRadians(double degrees)
        {
            return Math.PI / 180.0 * degrees;
        }

        static double RadiansToDegrees(double radians)
        {
            return 180.0 / Math.PI * radians;
        }
    }
    public class ExtrinsicSupport
    {
        public static bool IsDerivedCRS(string idString)
        {
            return idString.ToLower().Contains("conversion[");
        }
        public static bool IsFromAndToCRS(string idString)
        {
            return idString.Contains("=>");
        }
        public static bool GetFromAndToCRS(string idString, out string fromCRS, out
string toCRS)
        {
            fromCRS = "";
            toCRS = "";
            // Split at =>
            int arrowIndex = idString.IndexOf("=>");
            if (arrowIndex < 1)
            {
```

```
                return false;
            }
            fromCRS = idString.Substring(0, arrowIndex);
            toCRS = idString.Substring(arrowIndex + 2);
            return true;
        }
        public static string GetEPSGNumber(string wktString)
        {
            string epsgNumber = "";
            // look at end of WKT for WKT1 or WKT2 ID
            // "ID\["EPSG",\d+\]\]$" or "AUTHORITY\["EPSG",\"\d+\"\]\]$"
            Regex reID = new Regex("(id\\[\\\"epsg\\\",\\d+\\]\\]$)");
            Regex reAuthority = new
Regex("(authority\\[\\\"epsg\\\",\\\"\\d+\\\"\\]\\]$)");
            string thisMatch = "";
            MatchCollection matches;
            if ((matches = reID.Matches(wktString.ToLower())).Count > 0)
            {
                thisMatch = matches[0].Value;
            }
            else if ((matches = reAuthority.Matches(wktString.ToLower())).Count > 0)
            {
                thisMatch = matches[0].Value;
            }
            if (thisMatch != "")
            {
                Regex reNumber = new Regex("\\d+");
                matches = reNumber.Matches(thisMatch);
                if (matches.Count > 0)
                {
                    epsgNumber = matches[0].Value;
                }
            }
            return epsgNumber;
        }
        public static bool GetOriginParameters(string wktString, ref double[] origin)
        {
            // PARAMETER[\"Latitude of topocentric origin\",55,
            origin[0] = GetSignedDoubleInRe("parameter\\[\\\"latitude.+,-
?\\d+\\.?\\d*", wktString);
            origin[1] = GetSignedDoubleInRe("parameter\\[\\\"longitude.+,-
?\\d+\\.?\\d*", wktString);
            origin[2] = GetSignedDoubleInRe("parameter\\[\\\"[^\\]]*height.+,-
?\\d+\\.?\\d*", wktString);
            return (!double.IsNaN(origin[0]) && !double.IsNaN(origin[1]) &&
!double.IsNaN(origin[2]));
        }
        public static Positions.GeodeticPosition GetOriginParameters(string wktString)
        {
            // PARAMETER[\"Latitude of topocentric origin\",55,
            double lat = GetSignedDoubleInRe("parameter\\[\\\"latitude.+,-
```

```
?\\d+\\.?\\d*", wktString);
            double lon = GetSignedDoubleInRe("parameter\\[\\\"longitude.+,-
?\\d+\\.?\\d*", wktString);
            double h = GetSignedDoubleInRe("parameter\\[\\\"[^\\]]*height.+,-
?\\d+\\.?\\d*", wktString);
            if (!double.IsNaN(lat) && !double.IsNaN(lon) && !double.IsNaN(h))
            {
                return new Positions.GeodeticPosition(lat, lon, h);
            }
            return new Positions.NoPosition();
        }
        public static double GetSignedDoubleInRe(string reString, string inputString)
        {
            double result = double.NaN;
            Regex re = new Regex(reString);
            MatchCollection matches = re.Matches(inputString.ToLower());
            if (matches.Count > 0)
            {
                string thisMatch = matches[0].Value;
                re = new Regex("-?\\d+\\.?\\d*");
                matches = re.Matches(thisMatch);
                if (matches.Count > 0)
                {
                    result = double.Parse(matches[0].Value);
                }
            }
            return result;
        }
        public static Positions.GeodeticPosition GetPositionFromParameters(string
paramString)
        {
            // JSON encoded: {"lat": 12.345, "lon": -22.54, "h": 11.22}
            double lat = GetSignedDoubleInRe("\\\"lat\\\"\\s*:\\s*-?\\d+(\\.\\d*)?",
paramString);
            double lon = GetSignedDoubleInRe("\\\"lon\\\"\\s*:\\s*-?\\d+(\\.\\d*)?",
paramString);
            double h = GetSignedDoubleInRe("\\\"h\\\"\\s*:\\s*-?\\d+(\\.\\d*)?",
paramString);
            if (!double.IsNaN(lat) && !double.IsNaN(lon) && !double.IsNaN(h))
            {
                return new Positions.GeodeticPosition(lat, lon, h);
            }
            return new Positions.NoPosition();
        }
    }
}
```

Step 2: Positions

The Position class and its derivatives represent different styles of using three coordinate values to

designate a position in a three-dimensional space.

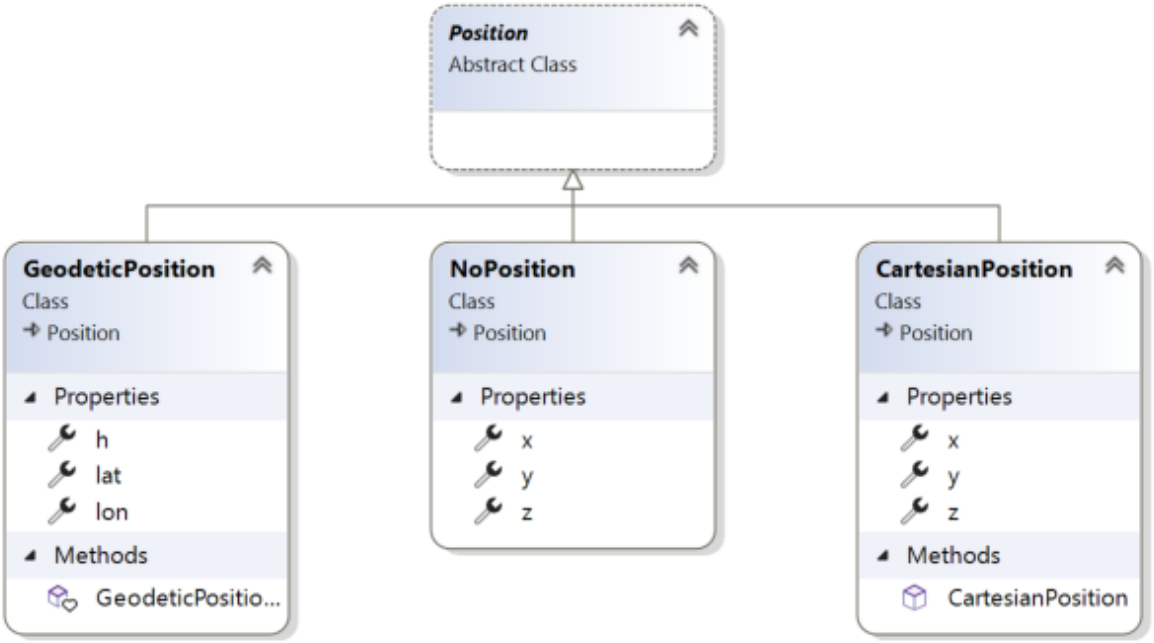

Figure 3. Positions

GeoPose 1.0 supports both a geodetic form and a Cartesian form. These forms are used in both frame transformations and orientation (rotation) transformations, both as quantities to be transformed and, in some cases, as a parameter of a family of transformations. Since some transformations are not possible, due to a mathematical singularity, unavailability of a transformation, or a runtime error in the transformation calculation, the NoPosition position is used as a "null" value. Each of the coordinates of the NoPosition are the IEEE 754 value NaN.

C# implementation:

Copyright and License (MIT)

```
// Implementation step: 2 - follows Extras.
// These classes define positions in a 3D frame using different conventions.

/// <summary>
/// The abstract root of the Position hierarchy.
/// <note>
/// Because these various ways to express Position share no underlying structure,
/// the abstract root class definition is simply an empty shell.
/// </note>
/// </summary>

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace Positions
```

```
{
    /// <summary>
    /// A specialization of Position for using two angles and a height for geodetic
positions.
    /// </summary>
    public class GeodeticPosition : Position
    {
        internal GeodeticPosition()
        {

        }
        public GeodeticPosition(double lat, double lon, double h)
        {
            this.lat = lat;
            this.lon = lon;
            this.h = h;
        }
        public static implicit operator GeodeticPosition(NoPosition noPosition)
        {
            return new GeodeticPosition(double.NaN, double.NaN, double.NaN);
        }

        /// <summary>
        /// A latitude in degrees, positive north of equator and negative south of
equator.
        /// The latitude is the angle between the plane of the equator and a plane
tangent to the ellipsoid at the given point.
        /// </summary>
        public double lat { get; set; } = double.NaN;
        /// <summary>
        /// A longitude in degrees, positive east of the prime meridian and negative
west of prime meridian.
        /// </summary>
        public double lon { get; set; } = double.NaN;
        /// <summary>
        /// A distance in meters, measured with respect to an implied (Basic) or
specified (Advanced) reference surface,
        /// postive opposite the direction of the force of gravity,
        /// and negative in the direction of the force of gravity.
        /// </summary>
        public double h { get; set; } = double.NaN;
    }
    /// <summary>
    /// A specialization of Position for geocentric positions.
    /// </summary>
    public class CartesianPosition : Position
    {
        protected CartesianPosition()
        {
        }
        public CartesianPosition(double x, double y, double z)
```

```
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        /// <summary>
        /// A coordinate value in meters, along an axis (x-axis) that typically has
origin at
        /// the center of mass, lies in the same plane as the y axis, and
perpendicular to the y axis,
        /// forming a right-hand coordinate system with the z-axis in the up
direction.
        /// </summary>
        public double x { get; set; } = double.NaN;
        /// <summary>
        /// A coordinate value in meters, along an axis (y-axis) that typically has
origin at
        /// the center of mass, lies in the same plane as the x axis, and
perpendicular to the x axis,
        /// forming a right-hand coordinate system with the z-axis in the up
direction.
        /// </summary>
        public double y { get; set; } = double.NaN;
        /// <summary>
        /// A coordinate value in meters, along the z-axis.
        /// </summary>
        public double z { get; set; } = double.NaN;
    }
    public class NoPosition : CartesianPosition
    {
        public NoPosition()
        {
            base.x = double.NaN;
            base.y = double.NaN;
            base.z = double.NaN;
        }
        /// <summary>
        /// A coordinate value in meters, along an axis (x-axis) that typically has
origin at
        /// the center of mass, lies in the same plane as the y axis, and
perpendicular to the y axis,
        /// forming a right-hand coordinate system with the z-axis in the up
direction.
        /// </summary>
        //public double x { get; set; } = double.NaN;
        /// <summary>
        /// A coordinate value in meters, along an axis (y-axis) that typically has
origin at
        /// the center of mass, lies in the same plane as the x axis, and
perpendicular to the x axis,
```

```
        /// forming a right-hand coordinate system with the z-axis in the up
direction.
        /// </summary>
        //public double y { get; set; } = double.NaN;
        /// <summary>
        /// A coordinate value in meters, along the z-axis.
        /// </summary>
        //public double z { get; set; } = double.NaN;
    }
    /// <summary>
    /// The abstract root of the Position hierarchy.
    /// <note>
    /// Because the various ways to express Position share no underlying structure,
    /// the class definition is simply an empty shell.
    /// </note>
    /// </summary>
    public abstract class Position
    {

    }
}
```

Step 3: Frame Transforms

The frame transform is the first of the two key elements of a GeoPose. It is a function that transforms a Position defined by three coordinates in a starting reference frame - the **outer** frame - to a Position in a destination reference frame - the ***inner** frame. The GeoPose 1.0 structure holds an explicit (Advanced form) or an implicit (Basic form) specification of the outer frame, the transformation, and the inner frame.

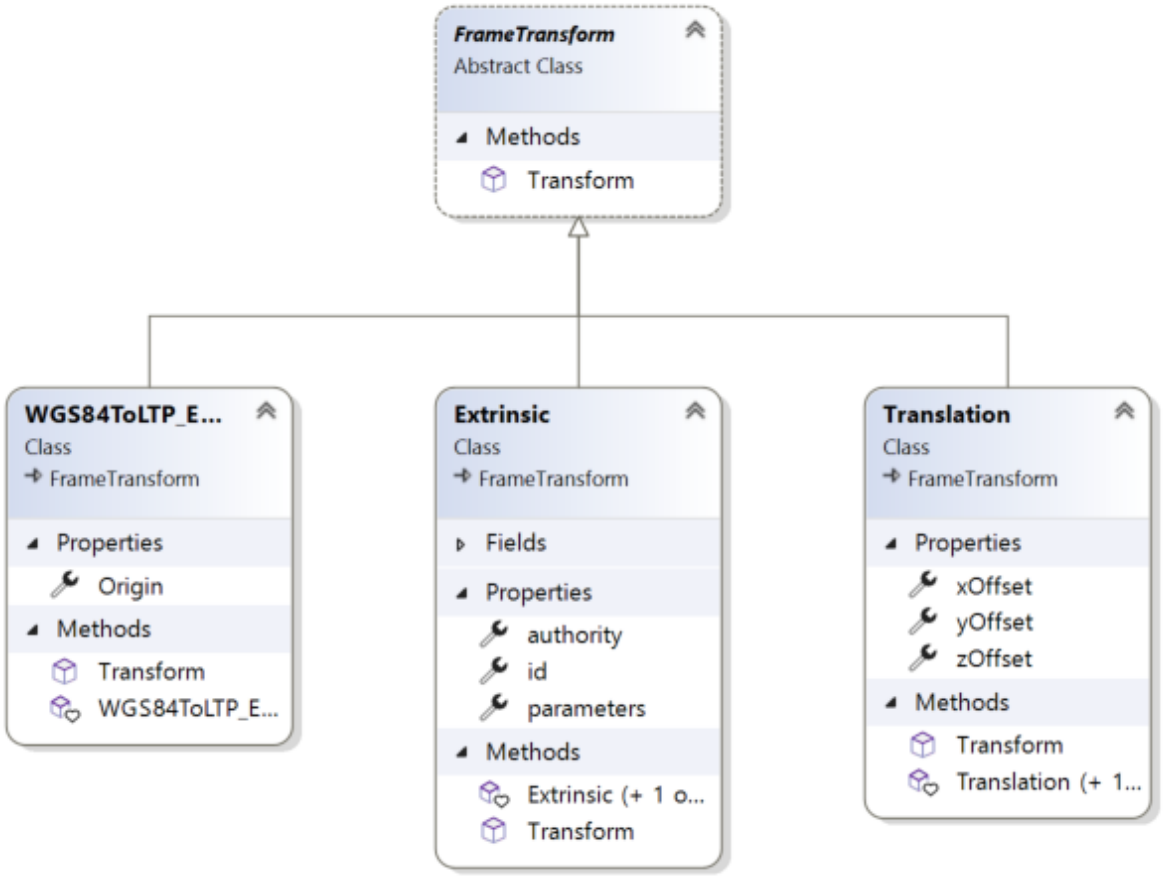

Figure 4. Frame Transform

The Basic form uses an implicit specification of an outer frame based on the WGS84 datum and geodetic coordinates, a transformation to a Cartesian tangent plane coordinate system in the inner frame. The outer frame is the EPSG 4979, the transformation is EPSG 9837, and the inner frame is EPSG 5819.

The Advanced form uses three strings - **authority**, **id**, and **parameters** to provide a linkage between the GeoPose structure and an external method of denoting either

- an outer frame (datum) and Position + transformation,
- an outer frame, and an inner frame with an implicit transformation, or
- an outer frame, a transformation, and an inner frame.

The usage of these three fields to provide the linkage **is determined by the software provider** since GeoPose is independent of the external organizations and correspondingly, the external organizations are not aware of GeoPose. One example linkage is provided in the included code, linking to the PROJ library for JavaScript (ProjJS).

The Local form uses an implicit Translation transformation between outer and inner frames, where Positions in both are expressed as Cartesian coordinates.

It is important to note that the GeoPose implementation not only contains the references needed to define outer frame, transformation, and inner frame but also must **implement** the transformation.

C# implementation:

Copyright and License (MIT)

```
// Implementation order: 3 - follows Position.
// These classes define transformations of a Position in one 3D frame to a Position in
another 3D frame.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using ProjNet.CoordinateSystems;
using ProjNet.CoordinateSystems.Transformations;
using GeoAPI.CoordinateSystems.Transformations;

using Positions;
using Support;

namespace FrameTransforms
{
    /// <summary>
    /// A FrameTransform is a generic container for information that defines mapping
between reference frames.
    /// Most transformation have a context with necessary ancillary information
    /// that parameterizes the transformation of a Position in one frame to a
corresponding Position is another.
    /// Such context may include, for example, some or all of the information that may
be conveyed in an ISO 19111 CRS specification
    /// or a proprietary naming, numbering, or modelling scheme as used by EPSG, NASA
Spice, or SEDRIS SRM.
    /// Subclasses of FrameTransform exist precisely to hold this context in
conjunction with code
    /// implementing a Transform function.
    /// <remark>
    /// </remark>
    /// </summary>
    public abstract class FrameTransform
    {
        public virtual Position Transform(Position point)
        {
            // The defualt is to apply the identity transformation
            Position result = point;
            return result;
        }
    }

    /// <summary>
    /// A FrameSpecification is a generic container for information that defines a
```

```
reference frame.
    /// <remark>
    /// A FrameSpecification can be abstracted as a Position:
    /// The origin of the coordinate system associated with the frame is a Position
and serves in that role
    /// in the Advanced GeoPose.
    /// The origin, is in fact the *only* distinguished Position associated with the
coodinate system.
    /// </remark>
    /// </summary>
    public class Extrinsic : FrameTransform
    {
        internal Extrinsic()
        {

        }
        public Extrinsic(string authority, string id, string parameters)
        {
            this.authority = authority;
            this.id = id;
            this.parameters = parameters;
        }
        /// <summary>
        /// The core function of a transformation is the implement a specific frame
transformation
        /// i.e. the transformation of a triple of point coordinates in the outer
frame to a triple of point coordinates in the inner frame.
        /// When this is not possible due to lack of an appropriate tranformation
procedure,
        /// the triple (NaN, NaN, NaN) [three IEEE 574 not-a-number vales] is
returned.
        /// Note that an "authority" is not necessarily a standards organization but
rather an entity that provides
        /// a register of some kind for a category of frame- and/or frame transform
specifications that is useful and stable enough
        /// for someone to implement transformation functions.
        /// An implementation need not implement all possbile transforms.
        /// </summary>
        /// <note>
        /// This would be a good element to implement as a set of plugin.
        /// </note>
        /// <param name="point"></param>
        /// <returns></returns>
        public override Position Transform(Position point)
        {
            string uri = authority.ToLower().Replace("//www.", "");
            if (uri == "https://proj.org" || uri == "https://osgeo.org")
            {
                CoordinateTransformationFactory ctfac = new
CoordinateTransformationFactory();
                string WKT = "PROJCRS[\"GeoPose LTP-ENU\", GEOGCS[\"WGS 84 (G873)\",
```

```
DATUM[\"World_Geodetic_System_1984_G873\", " +
                    "SPHEROID[\"WGS 84\",6378137,298.257223563,
AUTHORITY[\"EPSG\",\"7030\"]], AUTHORITY[\"EPSG\",\"1153\"]], " +
                    "PRIMEM[\"Greenwich\",0, AUTHORITY[\"EPSG\",\"8901\"]],
UNIT[\"degree\",0.0174532925199433, AUTHORITY[\"EPSG\",\"9122\"]], " +
                    "AUTHORITY[\"EPSG\",\"9054\"]] CONVERSION[\"Topocentric LTP-ENU\",
" +
                    "METHOD[\"Geographic/topocentric conversions\",
ID[\"EPSG\",9837]], " +
                    "PARAMETER[\"Latitude of topocentric origin\",55,
ANGLEUNIT[\"degree\",0.0174532925199433], ID[\"EPSG\",8834]], " +
                    "PARAMETER[\"Longitude of topocentric origin\",5,
ANGLEUNIT[\"degree\",0.0174532925199433], ID[\"EPSG\",8835]], " +
                    "PARAMETER[\"Ellipsoidal height of topocentric origin\",0,
LENGTHUNIT[\"metre\",1], ID[\"EPSG\",8836]], " +
                    "ID[\"EPSG\",15594]] CS[Cartesian,3], " +
                    "AXIS[\"topocentric East (U)\",east, ORDER[1],
LENGTHUNIT[\"metre\",1]], " +
                    "AXIS[\"topocentric North (V)\",north, ORDER[2],
LENGTHUNIT[\"metre\",1]], " +
                    "AXIS[\"topocentric height (W)\",up, ORDER[3],
LENGTHUNIT[\"metre\",1]] " +
                    "USAGE[ AREA[\"Planet Earth\"], BBOX[-90,-180,90,180]],
ID[\"GeoPose\",LTP-ENU]]";

                var from = GeographicCoordinateSystem.WGS84;
                var to =
ProjNet.CoordinateSystems.ProjectedCoordinateSystem.WGS84_UTM(30, true);

                // convert points from one coordinate system to another
                ProjNet.CoordinateSystems.Transformations.ICoordinateTransformation
trans = ctfac.CreateFromCoordinateSystems(from, to);

                ProjNet.Geometries.XY businessCoordinate = new ProjNet.Geometries.XY(-
0.127758, 51.507351);
                ProjNet.Geometries.XY searchLocationCoordinate = new
ProjNet.Geometries.XY(-0.142500, 51.539188);

                MathTransform mathTransform = trans.MathTransform;
                var businessLocation = mathTransform.Transform(businessCoordinate.X,
businessCoordinate.Y);
                var searchLocation =
mathTransform.Transform(searchLocationCoordinate.X, searchLocationCoordinate.Y);



                return noTransform;
            }
            else if (uri == "https://epsg.org")
            {
                return noTransform;
```

```
            }
            else if (uri == "https://iers.org")
            {
                return noTransform;
            }
            else if (uri == "https://naif.jpl.nasa.gov")
            {
                return noTransform;
            }
            else if (uri == "https://sedris.org")
            {
                return noTransform;
            }
            else if (uri == "https://iau.org")
            {
                return noTransform;
            }
            else if (uri == "https://geopose.io")
            {
                // extract pair of CS wkts or CS plus transformation from the id
string
                // get the point to transform from the parameters string, JSON-encoded
as
                //  {"lat": 12.345, "lon": -22.54, "h": 11.22}
                //  if contains "=>" then that splits the outer and inner specs
                //  else check as special case: ID[\"EPSG\",5819]]$ or
AUTHORITY[\"EPGS\",\"5819\"]]$
                if (Support.ExtrinsicSupport.IsDerivedCRS(id))
                {
                    //
                    string epsgNumber = Support.ExtrinsicSupport.GetEPSGNumber(id);
                    if (epsgNumber == null || epsgNumber == "")
                    {
                        return noTransform;
                    }
                    else if (epsgNumber == "5819")
                    {
                        // get lat0, lon0, h0
                        double[] origin = new double[3];
                        if (Support.ExtrinsicSupport.GetOriginParameters(id, ref
origin))
                        {
                            Positions.GeodeticPosition inPoint =
Support.ExtrinsicSupport.GetPositionFromParameters(this.parameters);
                            Positions.GeodeticPosition tangentPoint = new
Positions.GeodeticPosition(origin[0], origin[1], origin[2]);
                            Positions.CartesianPosition outPoint =
Support.LTP_ENU.GeodeticToEnu(inPoint, tangentPoint);
                        }
                    }
                    else
```

```
                    {
                        return noTransform;
                    }
                }
                else if (Support.ExtrinsicSupport.IsFromAndToCRS(id))
                {
                    string fromCRS = "";
                    string toCRS = "";
                    if (Support.ExtrinsicSupport.GetFromAndToCRS(id, out fromCRS, out
toCRS))
                    {

                        var cf = new CoordinateSystemFactory();
                        var f = new CoordinateTransformationFactory();
                        CoordinateSystem csIn = null;
                        try
                        {
                            csIn = cf.CreateFromWkt(fromCRS);
                        }
                        catch (Exception ex)
                        {
                            return noTransform;
                        }
                        CoordinateSystem csOut = null;
                        try
                        {
                            csOut = cf.CreateFromWkt(toCRS);
                        }
                        catch (Exception ex)
                        {
                            return noTransform;
                        }
                        //var cs3857 = cf.CreateFromWkt(wkt3857);

ProjNet.CoordinateSystems.Transformations.ICoordinateTransformation transform = null;
                        if (csIn != null && csOut != null)
                        {
                            Positions.GeodeticPosition inPoint =
Support.ExtrinsicSupport.GetPositionFromParameters(this.parameters);
                            transform = f.CreateFromCoordinateSystems(csIn, csOut);
                            double[] xyz = new double[3] {inPoint.lon, inPoint.lat,
inPoint.h}; // note lon, lat, h order
                            double[] XYZ = new double[3];
                            double[] ret = transform.MathTransform.Transform(xyz);
                            XYZ[0] = ret[0];
                            XYZ[1] = ret[1];
                            if (ret.Length == 2)
                            {
                                XYZ[2] = xyz[2];
                            }
                            else
```

```
                        {
                            XYZ[2] = ret[2];
                        }
                    }
                    else
                    {
                        Console.WriteLine("Coordinate transformation failed: ");
                    }
                }
                else
                {
                    Console.WriteLine("from or to CS unrecognized");
                }

            }
            else
            {
                Console.WriteLine("id string missing \"=>\".");
            }

            return new Positions.NoPosition();
        }
        return noTransform;
    }
    /// <summary>
    /// The name or identification of the definer of the category of frame
specification.
    /// A Uri that usually but not always points to a valid web address.
    /// </summary>
    public string authority { get; set; } = "";
    /// <summary>
    /// A string that uniquely identifies a frame type.
    /// The interpretation of the string is determined by the authority.
    /// </summary>
    public string id { get; set; } = "";
    /// <summary>
    /// A string that holds any parameters required by the authority to define a
frame of the given type as specified by the id.
    /// The interpretation of the string is determined by the authority.
    /// </summary>
    public string parameters { get; set; } = "";
    static Position noTransform = new NoPosition();
}
/// <summary>
/// A specialized specification of the WGS84 (EPSG 4326) geodetic frame to a local
tangent plane East, North, Up frame.
/// <remark>
/// The origin of the coordinate system associated with the frame is a Position -
the origin -
/// which is the *only* distinguished Position associated with the coodinate
system associated with the inner frame (range).
```

```
    /// </remark>
    /// </summary>
    public class WGS84ToLTP_ENU : FrameTransform
    {
        internal WGS84ToLTP_ENU()
        {

        }
        public WGS84ToLTP_ENU(GeodeticPosition origin)
        {
            this.Origin = origin;
        }
        public override Position Transform(Position point)
        {
            double east, north, up;
            Support.LTP_ENU.GeodeticToEnu(((GeodeticPosition)point).lat,
((GeodeticPosition)point).lon, ((GeodeticPosition)point).h,
                Origin.lat, Origin.lon, Origin.h, out east, out north, out up);
            CartesianPosition outPoint = new CartesianPosition(east, north, up);
            return outPoint;
        }

        /// <summary>
        /// A single geodetic position defines the tangent point for a transform to
LTP-ENU.
        /// </summary>
        public GeodeticPosition Origin { get; set; } = null;
    }

    // A simple translation frame transform.
    // The FrameTransform is created with an offset.
    // The Transform adds the offset ot an input Cartesian Position and
    // returns a Cartesian Position.
    public class Translation : FrameTransform
    {
        internal Translation()
        {
        }
        public Translation(double xOffset, double yOffset, double zOffset)
        {
            this.xOffset = xOffset;
            this.yOffset = yOffset;
            this.zOffset = zOffset;
        }
        public override Position Transform(Position point)
        {
            return new CartesianPosition(((CartesianPosition)point).x,
((CartesianPosition)point).y, ((CartesianPosition)point).z);
        }
        public double xOffset { get; set; } = 0.0;
        public double yOffset { get; set; } = 0.0;
```

```
        public double zOffset { get; set; } = 0.0;
    }
}
```

Step 4: Orientations

The Orientation is the second key GeoPose element. The Orientation is a rotational transformation that takes a (any) Position in the inner frame and rotates it to a new position. It is best thought of as a rotation of the inner frame.

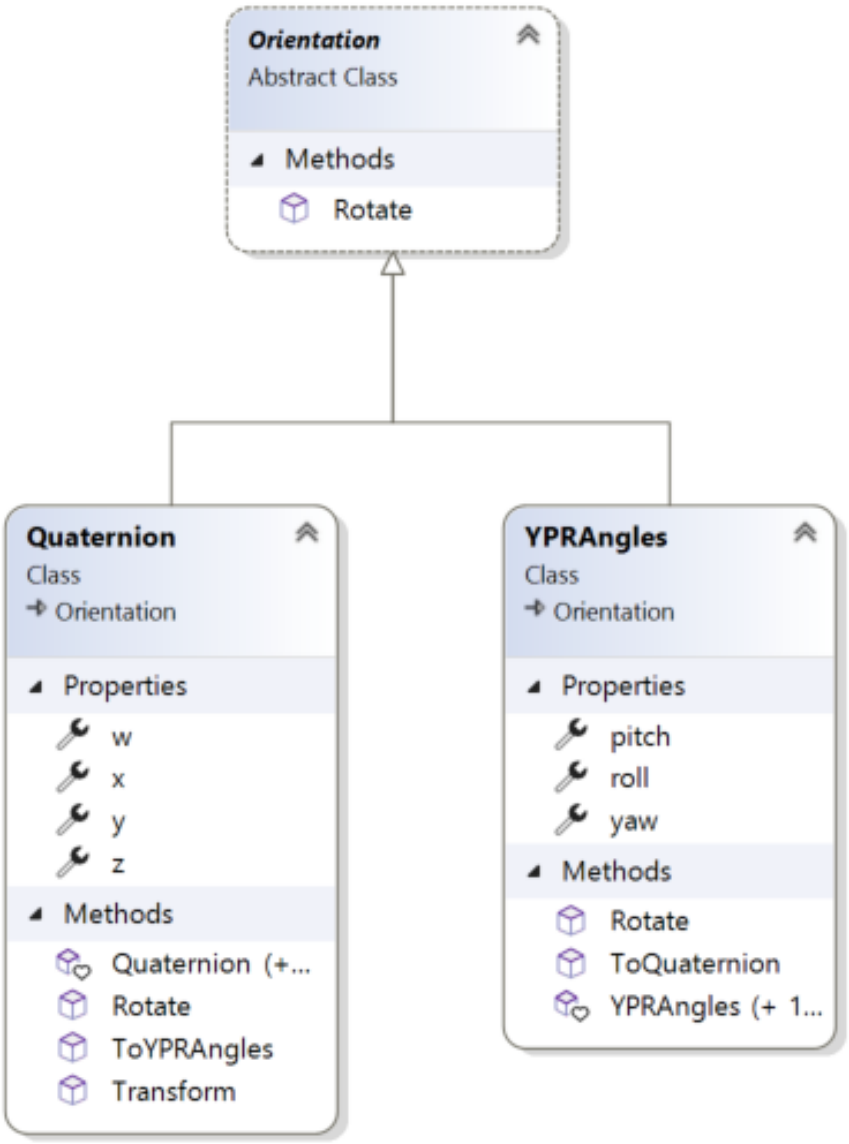

Figure 5. Orientations

There are several possible ways to specify a rotation.

One approach is to use consecutive rotations about each of the three axes. This is the easiest for human interpretation at a glance, but suffers from four difficulties, which may or may not outweigh the human-friendliness of successive rotations:

- there is an arbitrary choice of the order of axes about which to rotate,

- there is an arbitrary choice of whether the axes of rotation are the original unrotated axes or the new local axes are to be used after each rotation,
- there are singularities for rotation of a multiple of half of a circle, and
- interpolation of angular rotations is not uniform.

A second approach is to use a unit quaternion, which cannot be easily visualized but which offers good interpolation properties and an unambiguous interpretation.

As with the FrameTransform, Orientation classes must implement the actual rotational transformation.

Yaw, Pitch, Roll

yaw, pitch, and roll angles are one choice for a representation based on successive rotations about the z, y, and x axes, the axes being the local rotated axes after previous rotations.

Unit Quaternions

Unit quaternions have four components, the square root of the sum of the squares of which is 1.0.

C# implementation:


```
// Implementation order: 4 - follows FrameTransform.
// These classes define rotations of a 3D frame transforming a Position to a rotated
Position.

using System;

using Positions;

/// <summary>
/// The abstract root of the Orientation hierarchy.
/// <note>
/// An Orientation is a generic container for information that defines rotation within
a coordinate system associated with a reference frame.
/// An Orientation may have a specialized context with necessary ancillary information
/// that parameterizes the rotation.
/// Such context may include, for example, part of the information that may be
conveyed in an ISO 19111 CRS specification
/// or a proprietary naming, numbering, or modelling scheme as used by EPSG, NASA
Spice, or SEDRIS SRM.
/// Subclasses of Orientation exist precisely to hold this context in conjunction with
code
/// implementing a Rotate function.
/// </note>
/// </summary>
```

```
namespace Orientations
{
    /// <summary>
    /// The abstract root of the Orientation hierarchy.
    /// <note>
    /// An Orientation is a generic container for information that defines rotation
within a
    /// coordinate system associated with a reference frame.
    /// An Orientation may have a specialized context with necessary ancillary
information
    /// that parameterizes the rotation.
    /// Such context may include, for example, part of the information that
    /// may be conveyed in an ISO 19111 CRS specification
    /// or a proprietary naming, numbering, or modelling scheme as used by EPSG,
    /// NASA Spice, or SEDRIS SRM.
    /// Subclasses of Orientation exist precisely to hold this context in conjunction
with code
    /// implementing a Rotate function.
    /// </note>
    /// </summary>
    public abstract class Orientation
    {
        public virtual Position Rotate(Position point)
        {
            return point;
        }
    }
    /// <summary>
    /// A specialization of Orientation using Yaw, Pitch, and Roll angles in degrees.
    /// <remark>
    /// This style of Orientation is best for easy human interpretation.
    /// It suffers from some computational inefficiencies, awkward interpolation, and
singularities.
    /// </remark>
    /// </summary>
    public class YPRAngles : Orientation
    {
        // Prevent public use of empty constructor.
        internal YPRAngles()
        {

        }
        public YPRAngles(double yaw, double pitch, double roll)
        {
            this.yaw = yaw;
            this.pitch = pitch;
            this.roll = roll;
        }
        public override Position Rotate(Position point)
        {
            // convert to quaternion and use quaternion rotation
```

```
            Quaternion q = YPRAngles.ToQuaternion(this.yaw, this.pitch, this.roll);
            return Quaternion.Transform(point, q);
        }
        public static Quaternion ToQuaternion(double yaw, double pitch, double roll)
        {
            // GeoPose uses angles in degrees for human readability
            // Convert degrees to radians.
            yaw *= (Math.PI / 180.0);
            pitch *= (Math.PI / 180.0);
            roll *= (Math.PI / 180.0);

            double cosRoll = Math.Cos(roll * 0.5);
            double sinRoll = Math.Sin(roll * 0.5);
            double cosPitch = Math.Cos(pitch * 0.5);
            double sinPitch = Math.Sin(pitch * 0.5);
            double cosYaw = Math.Cos(yaw * 0.5);
            double sinYaw = Math.Sin(yaw * 0.5);

            double w = cosRoll * cosPitch * cosYaw + sinRoll * sinPitch * sinYaw;
            double x = sinRoll * cosPitch * cosYaw - cosRoll * sinPitch * sinYaw;
            double y = cosRoll * sinPitch * cosYaw + sinRoll * cosPitch * sinYaw;
            double z = cosRoll * cosPitch * sinYaw - sinRoll * sinPitch * cosYaw;

            double norm = Math.Sqrt(x * x + y * y + z * z + w * w);
            if (norm <= double.MinValue)
            {
                return new Quaternion(x, y, z, w);
            }
            return new Quaternion(x / norm, y / norm, z / norm, w / norm);
        }
        /// <summary>
        /// A left-right angle in degrees.
        /// </summary>
        public double yaw { get; set; } = double.NaN;
        /// <summary>
        /// An up-down angle in degrees.
        /// </summary>
        public double pitch { get; set; } = double.NaN;
        /// <summary>
        /// A side-to-side angle in degrees.
        /// </summary>
        public double roll { get; set; } = double.NaN;
    }
    /// <summary>
    /// A specialization of Orientation using a unit quaternion.
    /// </summary>
    /// <remark>
    /// This style of Orientation is best for computation.
    /// It is not easily interpreted or visualized by humans.
    /// </remark>
    public class Quaternion : Orientation
```

```
    {
        internal Quaternion()
        {

        }
        public Quaternion(double x, double y, double z, double w)
        {
            this.x = x;
            this.y = y;
            this.z = z;
            this.w = w;
        }
        public override Position Rotate(Position point)
        {
            return Quaternion.Transform(point, this);
        }
        public YPRAngles ToYPRAngles(Quaternion q)
        {
            YPRAngles yprAngles = new YPRAngles();

            // roll (x-axis rotation)
            double sinRollCosPitch = 2.0 * (q.w * q.x + q.y * q.z);
            double cosRollCosPitch = 1.0 - 2.0 * (q.x * q.x + q.y * q.y);
            yprAngles.roll = Math.Atan2(sinRollCosPitch, cosRollCosPitch) * (180.0 /
Math.PI); // in degrees

            // pitch (y-axis rotation)
            double sinPitch = Math.Sqrt(1.0 + 2.0 * (q.w * q.y - q.x * q.z));
            double cosPitch = Math.Sqrt(1.0 - 2.0 * (q.w * q.y - q.x * q.z));
            yprAngles.pitch = (2.0 * Math.Atan2(sinPitch, cosPitch) - Math.PI / 2.0) *
(180.0 / Math.PI); // in degrees

            // yaw (z-axis rotation)
            double sinYawCosPitch = 2.0 * (q.w * q.z + q.x * q.y);
            double cosYawCosPitch = 1.0 - 2.0 * (q.y * q.y + q.z * q.z);
            yprAngles.yaw = Math.Atan2(sinYawCosPitch, cosYawCosPitch) * (180.0 /
Math.PI); // in degrees

            return yprAngles;
        }
        public static Position Transform(Position inPoint, Quaternion rotation)
        {
            CartesianPosition point = (CartesianPosition)inPoint;
            double x2 = rotation.x + rotation.x;
            double y2 = rotation.y + rotation.y;
            double z2 = rotation.z + rotation.z;

            double wx2 = rotation.w * x2;
            double wy2 = rotation.w * y2;
            double wz2 = rotation.w * z2;
            double xx2 = rotation.x * x2;
```

```
            double xy2 = rotation.x * y2;
            double xz2 = rotation.x * z2;
            double yy2 = rotation.y * y2;
            double yz2 = rotation.y * z2;
            double zz2 = rotation.z * z2;

            return new CartesianPosition(
                point.x * (1.0f - yy2 - zz2) + point.y * (xy2 - wz2) + point.z * (xz2
+ wy2),
                point.x * (xy2 + wz2) + point.y * (1.0f - xx2 - zz2) + point.z * (yz2
- wx2),
                point.x * (xz2 - wy2) + point.y * (yz2 + wx2) + point.z * (1.0f - xx2
- yy2));
        }
        /// <summary>
        /// The x component.
        /// </summary>
        public double x { get; set; } = double.NaN;
        /// <summary>
        /// The y component.
        /// </summary>
        public double y { get; set; } = double.NaN;
        /// <summary>
        /// The z component.
        /// </summary>
        public double z { get; set; } = double.NaN;
        /// <summary>
        /// The w component.
        /// </summary>
        public double w { get; set; } = double.NaN;
    }
}
```

Step 5: Abstract GeoPose

The Abstract GeoPose is the root of the inheritance hierarchy. The distinction between GeoPose forms is determined by overriding implementations of the two elements of type FrameTransform and Orientation, respectively.

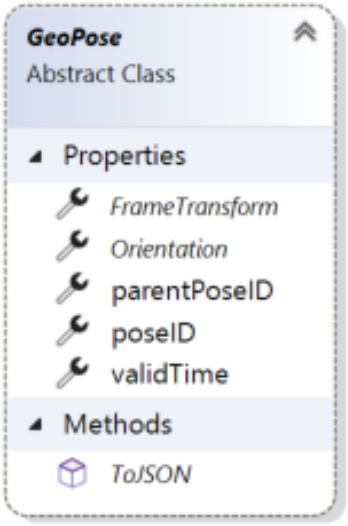

Figure 6. GeoPose

The three properties PoseID, parentPoseID, and validTime are additional properties not define by GeoPose 1.0 but allowed as additional properties in the serialized JSON data objects.

C# implementation:

Copyright and License (MIT)

```
// Implementation order: 5 - follows Orientation.
// This is the root of the GeoPose inheritance hierarchy.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Positions;
using FrameTransforms;
using Orientations;
using Extras;

namespace GeoPose
{
    /// <summary>
    /// The abstract root of the GeoPose Basic and Advanced classes.
    /// </summary>
    public abstract class GeoPose
    {
        // Optional and non-standard but conforming added property:
        //   an identifier unique within an application.
        public PoseID? poseID { get; set; } = null;

        // Optional and non-standard but conforming added property:
        //  a PoseID type identifier of another GeoPose in the direction of the root
of a pose tree.
        public PoseID? parentPoseID { get; set; } = null;

        // Optional and non-standard but conforming added property:
        //   a poseID identifier in the direction of the root of a pose tree.
        public UnixTime? validTime { get; set; } = null;
        public abstract FrameTransform FrameTransform { get; set; }
        public abstract Orientation Orientation { get; set; }
        public abstract string ToJSON(string indent);
    }
}
```

Step 6: Basic GeoPoses

Basic GeoPoses are a family where the FrameTransform is a transform from a WGS84 geodetic system to a local tangent plane, east-north-up inner frame.

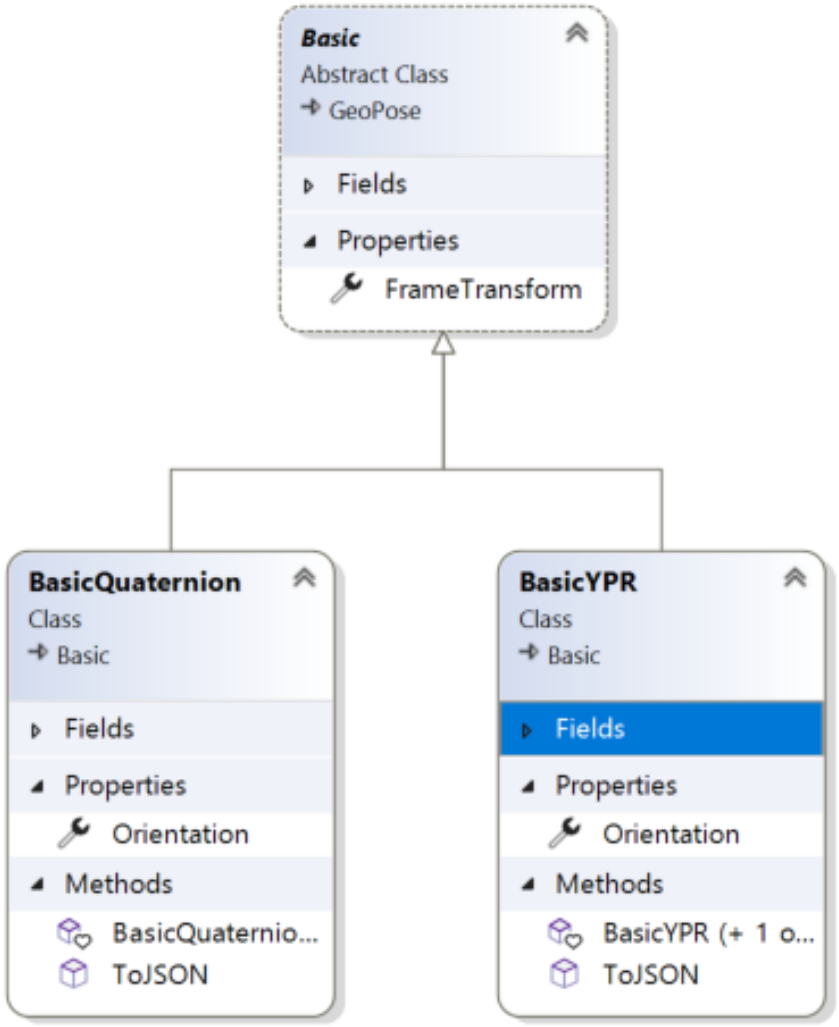

Figure 7. Basic

The two members of the Basic family are distinguished by the rotational transformation implementing the Orientation: BasicYPR and BasicQuaternion.

TypeScript implementation:

Copyright and License (MIT)

```
using System;
using System.Text;

using GeoPose;
using Positions;
using FrameTransforms;
using Orientations;
using Extras;

namespace Basic
{
    public abstract class Basic : GeoPose.GeoPose
    {
```

```
        /// <summary>
        /// A Position specified in spherical coordinates with height above a
reference surface -
        /// usually an ellipsoid of revolution or a gravitational equipotential
surface.
        /// </summary>
        private WGS84ToLTP_ENU _frameTransform = new WGS84ToLTP_ENU();
        public override FrameTransform FrameTransform
        {
            get
            {
                return _frameTransform;
            }
            set
            {
                if (value.GetType() == typeof(WGS84ToLTP_ENU))
                {
                    _frameTransform = (WGS84ToLTP_ENU)value;
                }
            }
        }
    }

    /// <summary>
    /// A Basic-YPR GeoPose.
    /// <remark>
    /// See the OGS GeoPose 1.0 standard for a full description.
    /// </remark>
    /// </summary>
    public class BasicYPR : Basic
    {
        internal BasicYPR()
        {

        }
        public BasicYPR(string id, GeodeticPosition tangentPoint, YPRAngles yprAngles)
        {
            this.poseID = new PoseID(id);
            this.FrameTransform = new WGS84ToLTP_ENU(tangentPoint);
            this.Orientation = yprAngles;
        }
        private YPRAngles _orientation = new YPRAngles();
        /// <summary>
        /// An Orientation specified as three rotations.
        /// </summary>
        public override Orientation Orientation
        {
            get
            {
                return _orientation;
            }
```

```
            set
            {
                if (value.GetType() == typeof(YPRAngles))
                {
                    _orientation = (YPRAngles)value;
                }
            }
        }
        /// <summary>
        /// Return a string containing Json encoding of a Basic-YPR GeoPose
        /// </summary>
        public override string ToJSON(string indent = "")
        {
            StringBuilder sb = new StringBuilder();
            if (FrameTransform != null && Orientation != null)
            {
                sb.Append("{\r\n\t\t" + indent);
                if (validTime != null && validTime.timeValue != String.Empty)
                {
                    sb.Append("\"validTime\": " + validTime.timeValue + ",\r\n" +
indent + "  ");
                }
                if (poseID != null && poseID.id != String.Empty)
                {
                    sb.Append("\"poseID\": \"" + poseID.id + "\",\r\n" + indent + "
");
                }
                if (parentPoseID != null && parentPoseID.id != String.Empty)
                {
                    sb.Append("\"parentPoseID\": \"" + parentPoseID.id + "\",\r\n" +
indent + "  ");
                }
                sb.Append("\"position\": {\r\n\t\t\t" + indent + "\"lat\": " +
((WGS84ToLTP_ENU)FrameTransform).Origin.lat + ",\r\n\t\t\t" + indent +
                    "\"lon\": " + ((WGS84ToLTP_ENU)FrameTransform).Origin.lon +
",\r\n\t\t\t" + indent +
                    "\"h\":   " + ((WGS84ToLTP_ENU)FrameTransform).Origin.h);
                sb.Append("\r\n\t\t" + indent + "},");
                sb.Append("\r\n\t\t" + indent);
                sb.Append("\"angles\": {\r\n\t\t\t" + indent + "\"yaw\":   " +
((YPRAngles)Orientation).yaw + ",\r\n\t\t\t" + indent +
                    "\"pitch\": " + ((YPRAngles)Orientation).pitch + ",\r\n\t\t\t" +
indent +
                    "\"roll\":  " + ((YPRAngles)Orientation).roll);
                sb.Append("\r\n\t\t" + indent + "}");
                sb.Append("\r\n\t" + indent + "}");
            }
            return sb.ToString();
        }
    }
```

```
    /// <summary>
    /// A Basic-Quaternion GeoPose.
    /// <remark>
    /// See the OGS GeoPose 1.0 standard for a full description.
    /// </remark>
    /// </summary>
    public class BasicQuaternion : Basic
    {
        internal BasicQuaternion()
        {
        }
        public BasicQuaternion(string id, GeodeticPosition tangentPoint, Quaternion
quaternion)
        {
            this.poseID = new PoseID(id);
            this.FrameTransform = new WGS84ToLTP_ENU(tangentPoint);
            this.Orientation = quaternion;
        }

        /// <summary>
        /// An Orientation specified as a unit quaternion.
        /// </summary>
        private Quaternion _orientation = new Quaternion();
        public override Orientation Orientation
        {
            get
            {
                return _orientation;
            }
            set
            {
                if (value.GetType() == typeof(Quaternion))
                {
                    _orientation = (Quaternion)value;
                }
            }
        }

        /// <summary>
        /// This function returns a Json encoding of a Basic-Quaternion GeoPose
        /// </summary>
        public override string ToJSON(string indent = "")
        {
            StringBuilder sb = new StringBuilder();
            if (((WGS84ToLTP_ENU)FrameTransform).Origin != null && Orientation !=
null)
            {
                sb.Append("{\r\n\t\t" + indent);
                if (validTime != null && validTime.timeValue != String.Empty)
                {
                    sb.Append("\"validTime\": " + validTime.timeValue + ",\r\n" +
```

```
indent + "  ");
                }
                if (poseID != null && poseID.id != String.Empty)
                {
                    sb.Append("\"poseID\": \"" + poseID.id + "\",\r\n" + indent + "
");
                }
                if (parentPoseID != null && parentPoseID.id != String.Empty)
                {
                    sb.Append("\"parentPoseID\": \"" + parentPoseID.id + "\",\r\n" +
indent + "  ");
                }
                sb.Append("\"position\": {\r\n\t\t\t" + indent + "\"lat\": " +
                    ((WGS84ToLTP_ENU)FrameTransform).Origin.lat + ",\r\n\t\t\t" +
indent +
                    "\"lon\": " + ((WGS84ToLTP_ENU)FrameTransform).Origin.lon +
                    ",\r\n\t\t\t" + indent +
                    "\"h\":   " + ((WGS84ToLTP_ENU)FrameTransform).Origin.h);
                sb.Append("\r\n\t\t" + indent + "},");
                sb.Append("\r\n\t\t" + indent);
                sb.Append("\"angles\": {\r\n\t\t\t" + indent + "\"x\":   " +
((Quaternion)Orientation).x + ",\r\n\t\t\t" + indent +
                    "\"y\": " + ((Quaternion)Orientation).y + ",\r\n\t\t\t" + indent +
                    "\"z\": " + ((Quaternion)Orientation).z + ",\r\n\t\t\t" + indent +
                    "\"w\":  " + ((Quaternion)Orientation).w);
                sb.Append("\r\n\t\t" + indent + "}");
                sb.Append("\r\n\t" + indent + "}");
            }
            return sb.ToString();
        }
    }
}
```

Step 7: Advanced GeoPose

The Advanced GeoPose provides an interface to external repositories and libraries supporting

- definition of coordinate reference systems
- definition of coordinate systems
- transformations between reference frames

This interface is defined in the Advanced class by the creation of the Extrinsic FrameTransform. The Extrinsic FrameTransform is a contained for a three element interface, each a string, and named **authority**, **id**, and **parameters**. The usage of the interface is outside the scope of the GeoPose 1.0 standard. It can vary between external sources and the **final definition and interface protocol is determined by the implementer of the Advanced class**.

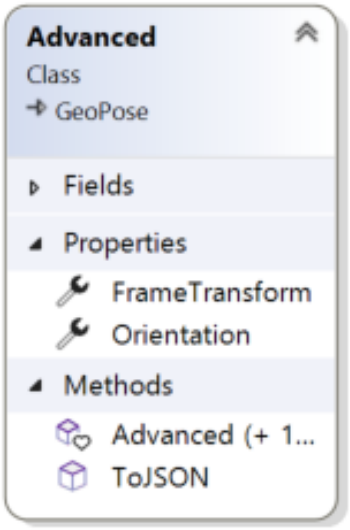

Figure 8. The Advanced Class

The Advanced class should implement each of the frame transforms, where validity is determined by the implementer. Presentation of an invalid frame specification via the Extrinsic interface should result in a returned NoPosition value. Note that these are suggestions to the implementer, not requirements.

C# implementation:

Copyright and License (MIT)

```
using System;
using System.Text;

using GeoPose;
using Extras;
using FrameTransforms;
using Orientations;

namespace Advanced
{
    /// <summary>
    /// Advanced GeoPose.
    /// </summary>
    public class Advanced : GeoPose.GeoPose
    {
        internal Advanced()
        {

        }
        public Advanced(PoseID poseID, Extrinsic frameTransform, Quaternion
orientation)
        {
            this.poseID = poseID;
            FrameTransform = frameTransform;
```

```
            Orientation = orientation;
        }

        /// <summary>
        /// A Frame Specification defining a frame with associated coordinate system
whose Position is the origin.
        /// </summary>
        private Extrinsic _frameTransform = new Extrinsic();
        public override FrameTransform FrameTransform
        {
            get
            {
                return _frameTransform;
            }
            set
            {
                if (value.GetType() == typeof(Extrinsic))
                {
                    _frameTransform = (Extrinsic)value;
                }
                // else throw expected extrinsic exception
            }
        }
        /// <summary>
        /// An Orientation specified as a unit quaternion.
        /// </summary>
        private Quaternion _orientation = new Quaternion();
        public override Orientation Orientation
        {
            get
            {
                return _orientation;
            }
            set
            {
                if (value.GetType() == typeof(Quaternion))
                {
                    _orientation = (Quaternion)value;
                }
            }
        }
        /// <summary>
        /// Milliseconds of Unix time ticks (optional).
        /// </summary>
        //public long? ValidTime { get; set; } = 0;
        /// <summary>
        /// This function returns a Json encoding of an Advanced GeoPose
        /// </summary>
        public override string ToJSON(string indent)
        {
            StringBuilder sb = new StringBuilder();
```

```
            sb.Append("{\r\n" + indent + "  ");
            if (validTime != null && validTime.timeValue != String.Empty)
            {
                sb.Append("\"validTime\": " + validTime.timeValue + ",\r\n" + indent +
"  ");
            }
            if (poseID != null && poseID.id != String.Empty)
            {
                sb.Append("\"poseID\": \"" + poseID.id + "\",\r\n" + indent + "  ");
            }
            if (parentPoseID != null && parentPoseID.id != String.Empty)
            {
                sb.Append("\"parentPoseID\": \"" + parentPoseID.id + "\",\r\n" +
indent + "  ");
            }
            sb.Append("\"frameSpecification\":\r\n" + indent + "  " + "{\r\n" + indent
+ "    \"authority\": \"" +
                ((Extrinsic)FrameTransform).authority.Replace("\"", "\\\"") +
"\",\r\n" + indent + "    \"id\": \"" +
                ((Extrinsic)FrameTransform).id.Replace("\"", "\\\"") + "\",\r\n" +
indent + "    \"parameters\": \"" +
                ((Extrinsic)FrameTransform).parameters.Replace("\"", "\\\"") +
"\"\r\n" + indent + "  },\r\n" + indent + "  ");
            sb.Append("\"quaternion\":\r\n" + indent + "  {\r\n" + indent + "
\"x\":" + ((Quaternion)Orientation).x + ",\"y\":" +
                ((Quaternion)Orientation).y + ",\"z\":" +
                ((Quaternion)Orientation).z + ",\"w\":" +
                ((Quaternion)Orientation).w);
            sb.Append("\r\n" + indent + "  }\r\n" + indent + "}\r\n");
            return sb.ToString();
        }
    }
}
```

Step 8: Local [Geo]Pose

The Local GeoPose is in essence, an implementation of the pose concept from computer graphics. It can be implemented as an Advanced geoPose but a lightweight implementation for operations within a local Cartesian coordinate system is often useful. I hope that a future version of OGC GeoPose has something like the Local form. Until then, there are three alternatives

- use a non-standard serialized form,
- serialize this class structure as an Advanced and compliant data object, or
- rely on the Advanced form.

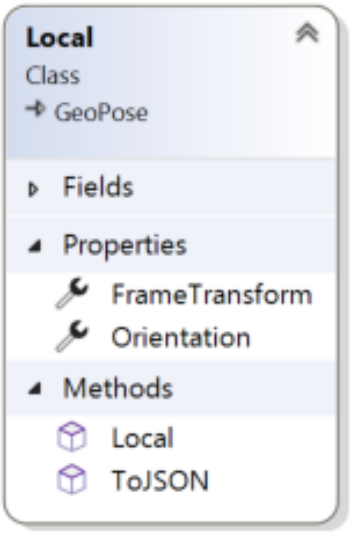

Figure 9. Local

The FrameTransform supported by the Local GeoPose is Translation.

C# implementation:

Copyright and License (MIT)

```
// Implemention order: 8 -a useful GeoPose for working within a local Cartesian (i.e.
engineering) frame.
// Local can be expressed as an Advanced form, but the Advanced form is more complex
and this implementation is a shortcut.

using System;
using System.Text;

using GeoPose;
using Positions;
using FrameTransforms;
using Orientations;
using Extras;

/// <summary>
/// A derived pose within an engineering CRS with a Cartesian coordinate system.
/// This form is the closest to the classical computer graphics pose concept.
/// <remark>
/// Not (yet) part of the OGC GeoPose standard and not backwards-compatible.
/// Useful when operating within a local Cartesian frame defined by a Basic (or other)
GeoPose.
/// </remark>
/// </summary>
public class Local : GeoPose.GeoPose
{
    public Local(string id, Translation frameTransform, Orientation quaternion)
    {
        this.poseID = new PoseID(id);
```

```
        this.FrameTransform = frameTransform;
        this.Orientation = quaternion;
    }
    /// <summary>
    /// The xOffset, yOffset, zOffset from the origin of the rotated inner frame of a
"parent" GeoPose.
    /// </summary>
    private Translation _frameTransform = new Translation();
    public override FrameTransform FrameTransform
    {
        get
        {
            return _frameTransform;
        }
        set
        {
            if (value.GetType() == typeof(Translation))
            {
                _frameTransform = (Translation)value;
            }
            // else throw expected translation exception
        }
    }
    /// <summary>
    /// Local uses the yaw, pitch, roll orientation.
    /// </summary>
    private YPRAngles _orientation = new YPRAngles();
    /// <summary>
    /// An Orientation specified as three rotations.
    /// </summary>
    public override Orientation Orientation
    {
        get
        {
            return _orientation;
        }
        set
        {
            if (value.GetType() == typeof(YPRAngles))
            {
                _orientation = (YPRAngles)value;
            }
            // else throw expected YPRAngles exception
        }
    }
    /// <summary>
    /// This function returns a Json encoding of a Basic-YPR GeoPose
    /// </summary>
    public override string ToJSON(string indent = "")
    {
        StringBuilder sb = new StringBuilder();
```

```
        sb.Append("{\r\n  ");
        if (validTime != null && validTime.timeValue != String.Empty)
        {
            sb.Append("\"validTime\": " + validTime.timeValue + ",\r\n" + indent + "
");
        }
        if (poseID != null && poseID.id != String.Empty)
        {
            sb.Append("\"poseID\": \"" + poseID.id + "\",\r\n" + indent + "  ");
        }
        if (parentPoseID != null && parentPoseID.id != String.Empty)
        {
            sb.Append("\"parentPoseID\": \"" + parentPoseID.id + "\",\r\n" + indent +
"  ");
        }
        sb.Append("\"position\": \r\n  {\r\n    " + "\"x\": " +
((Translation)FrameTransform).xOffset + ",\r\n    " +
            "\"y\": " + ((Translation)FrameTransform).yOffset + ",\r\n    " +
            "\"z\":   " + ((Translation)FrameTransform).zOffset);
        sb.Append("\r\n  " + "},");
        sb.Append("\r\n  ");
        sb.Append("\"angles\": \r\n  {\r\n    " + "\"yaw\":   " +
((YPRAngles)Orientation).yaw + ",\r\n    " +
            "\"pitch\": " + ((YPRAngles)Orientation).pitch + ",\r\n    " +
            "\"roll\":  " + ((YPRAngles)Orientation).roll);
        sb.Append("\r\n  " + "}");
        sb.Append("\r\n" + "}\r\n");
        return sb.ToString();
    }
}
```

Example

The following example uses each of the GeoPose forms to show different pose relationships can be modelled.

The example starts out with a helper class to display the current pose status of a "Space Train" that is somewhere in interplanetary space, possibly headed to Mars. Then there is a check of PROJ and finally the model of the Space Train itself.

C#

Copyright and License (MIT)

```
import { stdin as input } from 'node:process';
import * as proj4 from 'proj4';
import * as GeoPose from './GeoPose'
import * as Position from './Position';
```

```
import * as Orientation from './Orientation';
import * as LTPENU from './WGS84ToLTPENU';
import * as Basic from './Basic';
import * as Advanced from './Advanced';
import * as Local from './Local';
import * as FrameTransform from './FrameTransform';
import * as Extras from './Extras';

class Display {
    public static Output(spaceTrain: GeoPose.GeoPose, trainWagons: GeoPose.GeoPose[],
trainPassengers: GeoPose.GeoPose[]): void {
        console.log("\r\n========== Space Train at Local Clock UNIX Time " +
spaceTrain.validTime.toString() + "==========\r\n");
        console.log(spaceTrain.toJSON());
        trainWagons.forEach(function (wagon) {

            console.log("=-=-=-=-=- Wagon -=-=-=-=-=: " + wagon.poseID.id.substring(1
+ wagon.poseID.id.lastIndexOf('/')) + "\r\n");
            console.log(wagon.toJSON());
            trainPassengers.forEach(function (passenger) {

                if (passenger.parentPoseID.id == wagon.poseID.id) {
                    console.log("---------- Passenger ----------: " +
passenger.poseID.id.substring(1 + passenger.poseID.id.lastIndexOf('/')) + "\r\n");
                    console.log(passenger.toJSON());
                }
            })
        })
    }
}

//  - Verify that PROJ is configured and working
console.log("========== Checking PROJ ==========");
//    Source coordinates will be in Longitude/Latitude, WGS84
var source = proj4.Proj('EPSG:4326');
//    Destination coordinates in meters, global spherical mercators projection
var dest = proj4.Proj('EPSG:3785');

//  - Transform point coordinates
var p = proj4.toPoint([-76.0, 45.0, 11.0]);
let q = proj4.transform(source, dest, p);
let r = proj4.transform(dest, source, q);
console.log("X : " + p.x + " \nY : " + p.y + " \nZ : " + p.z);
console.log("X : " + q.x + " \nY : " + q.y + " \nZ : " + q.z);
console.log("X : " + r.x + " \nY : " + r.y + " \nZ : " + r.z);

let d = new LTPENU.LTP_ENU();
let from = new Position.GeodeticPosition(-1.0, 52.0, 15.0);
let origin = new Position.GeodeticPosition(-1.00005, 52.0, 15.3);
let to = new Position.CartesianPosition(0, 0, 0);
d.GeodeticToEnu(from, origin, to);
```

```
console.log('from: lat: ' + from.lat.toString() + " lon: " + from.lon.toString() + "
h: " + from.h.toString());
console.log('  to: x: ' + to.x.toString() + " y: " + to.y.toString() + " z: " +
to.z.toString());

//  - Display some example GeoPoses
console.log("========== Example GeoPoses ==========");
let myYPRLocal = new Basic.BasicYPR("OS_GB: BasicYPR",
    new Position.GeodeticPosition(51.5, -1.5, 12.3),
    new Orientation.YPRAngles(1, 2, 3));
let json = myYPRLocal.toJSON();
console.log(json);
let myQLocal = new Basic.BasicQuaternion("OS_GB: BasicQ",
    new Position.GeodeticPosition(51.5, -1.5, 23.4),
    new Orientation.Quaternion(0.1, 0.2, 0.3, 1.0));
json = myQLocal.toJSON();
console.log(json);
let myALocal = new Advanced.Advanced("OS_GB: Advanced",
    new FrameTransform.Extrinsic("epsg", "5819", "[1.5, -1.5, 23.4]"),
    new Orientation.Quaternion(0.1, 0.2, 0.3, 1.0));
json = myALocal.toJSON();
console.log(json);
let myLLocal = new Local.Local("OS_GB: Local",
    new FrameTransform.Translation(9.0, 8.7, 7.6),
    new Orientation.YPRAngles(1, 2, 3));
json = myLLocal.toJSON();
console.log(json);

//  - A "Space Train" example with interpose linkages
console.log("========== Space Train ==========");
// Create Mars Express in the current International Celestial Reference Frame ICRF2
let marsExpress = new Advanced.Advanced("https://example.com/nodes/MarsExpress/1",
    new FrameTransform.Extrinsic("https://www.iers.org/",
        "icrf3",
        "{\"x\": 1234567890.9876,\"y\": 2345678901.8765, \"z\": 3456789012.7654}"),
    new Orientation.Quaternion(0, 0, 0, 1));
marsExpress.validTime = 1674767748003;

//  - Create four 10 m long wagons with identical seat layouts in frames local to Mars
Express
//    and remember them in a wagons array
let wagons: Local.Local[] = [];
let wagon1 = new Local.Local("https://example.com/nodes/MarsExpress/1/Wagons/1",
    new FrameTransform.Translation(2.2, 0.82, -7.0),
    new Orientation.YPRAngles(0.2, 0.0, 23.0));
wagon1.parentPoseID = marsExpress.poseID;
wagons.push(wagon1);
let wagon2 = new Local.Local("https://example.com/nodes/MarsExpress/1/Wagons/2",
    new FrameTransform.Translation(12.2, 0.78, -7.0),
    new Orientation.YPRAngles(0.2, 0.0, 23.0));
wagon2.parentPoseID = marsExpress.poseID;
```

```
wagons.push(wagon2);
let wagon3 = new Local.Local("https://example.com/nodes/MarsExpress/1/Wagons/3",
    new FrameTransform.Translation(22.5, 0.77, -7.0),
    new Orientation.YPRAngles(0.2, 0.0, 23.0));
wagon3.parentPoseID = marsExpress.poseID;
wagons.push(wagon3);
let wagon4 = new Local.Local("https://example.com/nodes/MarsExpress/1/Wagons/4",
    new FrameTransform.Translation(33.2, 0.74, -7.0),
    new Orientation.YPRAngles(0.2, 0.0, 23.0));
wagon4.parentPoseID = marsExpress.poseID;
wagons.push(wagon4);

//  - Create passengers from the Cryptography Example Family (Alice, Bob, Carol, and
Charlie)
//    in wagons 1 and 3 in local frames local to specific wagons and
//    remember them in a passenger list
let passengers: GeoPose.GeoPose[] = [];

//  - Alice is a clever thinker who has many questions and good ideas
let Alice = new
Local.Local("https://example.com/nodes/MarsExpress/1/Passengers/Alice", new
FrameTransform.Translation(2.2, 0.8, -7.0), new Orientation.YPRAngles(180.0, 1.0,
0.0));
Alice.parentPoseID = wagon1.poseID;
passengers.push(Alice);

//  - Bob is a nice fellow who guided us towrd the frame transform in the early days
let Bob = new Local.Local("https://example.com/nodes/MarsExpress/1/Passengers/Bob",
new FrameTransform.Translation(2.0, 0.8, -6.0), new Orientation.YPRAngles(180.0, 2.0,
0.0));
Bob.parentPoseID = wagon1.poseID;
passengers.push(Bob);

//  - Carol thinks that the Local GeoPose is needed and should be added to version
1.1.0
let Carol = new
Local.Local("https://example.com/nodes/MarsExpress/1/Passengers/Carol", new
FrameTransform.Translation(-5.0, 0.82, 6.0), new Orientation.YPRAngles(-2.0, 1.5,
0.0));
Carol.parentPoseID = wagon3.poseID;
passengers.push(Carol);

//  - Charlie is one of Carol's multiple personalities. Charlie does not believe in
using any GeoPose not in the 1.0.0 standard
let Charlie =
    new Advanced.Advanced("https://charlie.com",
        new FrameTransform.Extrinsic(
            "https://ogc.org",
            "PROJCRS[\"GeoPose
Local\",+GEOGCS[\"None)\"]+CS[Cartesian,3],+AXIS[\"x\",,ORDER[1],LENGTHUNIT[\"metre\",
1]],+AXIS[\"y\",,ORDER[2],LENGTHUNIT[\"metre\",1]],+AXIS[\"z\",,ORDER[3],LENGTHUNIT[\"
```

```
metre\",1]]+USAGE[AREA[\"+/-1000 m\"],BBOX[-1000,-
1000,1000,1000],ID[\"GeoPose\",Local]]",
            "{\"x\": 1234567890.9876,\"y\": 2345678901.8765, \"z\":
3456789012.7654}"),
        new Orientation.Quaternion(0.0174509, 0.0130876, -0.0002284, 0.9997621));
Charlie.parentPoseID = wagon3.poseID;

//  - Charlie is going to do something time-dependent so we need to timestamp the
current info
Charlie.validTime = marsExpress.validTime; // Use the Mars Express local clock
passengers.push(Charlie);

//  - Display the pose tree
Display.Output(marsExpress, wagons, passengers);

//  - After a minute, the Charlie personality decides that he must split from the
Carol personality and
//    moves to the same seat in wagon 4.
//    Charlie's clock has an error of 327 millisecond with respect to marsExpress'
clock
marsExpress.validTime = 1674767748003 + 60 * 1000;
Charlie.parentPoseID = wagon4.poseID;
//  - Charlie moved so we need to update his clock
Charlie.validTime = marsExpress.validTime + 327; // Use the Mars Express local clock

//  - Display new pose tree
Display.Output(marsExpress, wagons, passengers);

// - Done
console.log("Enter to exit")
input.read();
```

Results

Here is the console log when running the Space Train example with node.js:

```
========== Checking PROJ ==========

X : -76
Y : 45
Z : 11
X : -8460281.300288793
Y : 5621521.486192066
Z : 11
X : -76.00000000000001
Y : 44.99999999999999
Z : 11
from: lat: -1 lon: 52 h: 15
  to: x: -4.058880592738845e-10 y: 5.528743791653243 z: -0.3000024121489482
```

```
========== Example GeoPoses ==========
{
  "poseID": "OS_GB: BasicYPR",
  "position":
  {
    "lat": 51.5,
    "lon": -1.5,
    "h":   12.3
  },
  "angles":
  {
    "yaw":   1,
    "pitch": 2,
    "roll":  3
  }
}
{
  "poseID": "OS_GB: BasicQ",
  "position":
  {
    "lat": 51.5,
    "lon": -1.5,
    "h":   23.4
  },
  "quaternion":
  {
    "x":   0.1,
      "y": 0.2,
      "z": 0.3,
      "w":  1
  }
}
{
  "poseID": "OS_GB: Advanced",
  "frameSpecification":
  {
    "authority": "epsg",
    "id": "5819",
    "parameters": "[1.5, -1.5, 23.4]"
  },
  "quaternion":
  {
    "x":0.1,"y":0.2,"z":0.3,"w":1
  }
}

{
  "poseID": "OS_GB: Local",
  "position":
  {
```

```
    "x": 9,
    "y": 8.7,
    "z":   7.6
  },
  "angles":
  {
    "yaw":   1,
    "pitch": 2,
    "roll":  3
  }
}

========== Space Train ==========

========== Space Train at Local Clock UNIX Time 1674767748003==========

{
  "validTime": 1674767748003,
  "poseID": "https://example.com/nodes/MarsExpress/1",
  "frameSpecification":
  {
    "authority": "https://www.iers.org/",
    "id": "icrf3",
    "parameters": "{\"x": 1234567890.9876,"y": 2345678901.8765, "z": 3456789012.7654}"
  },
  "quaternion":
  {
    "x":0,"y":0,"z":0,"w":1
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 1

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/1",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 2.2,
    "y": 0.82,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}

---------- Passenger ----------: Alice
```

```
{
  "poseID": "https://example.com/nodes/MarsExpress/1/Passengers/Alice",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/1",
  "position":
  {
    "x": 2.2,
    "y": 0.8,
    "z":   -7
  },
  "angles":
  {
    "yaw":   180,
    "pitch": 1,
    "roll":  0
  }
}

---------- Passenger ----------: Bob

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Passengers/Bob",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/1",
  "position":
  {
    "x": 2,
    "y": 0.8,
    "z":   -6
  },
  "angles":
  {
    "yaw":   180,
    "pitch": 2,
    "roll":  0
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 2

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/2",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 12.2,
    "y": 0.78,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
```

```
    "pitch": 0,
    "roll":  23
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 3

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/3",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 22.5,
    "y": 0.77,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}

---------- Passenger ----------: Carol

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Passengers/Carol",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/3",
  "position":
  {
    "x": -5,
    "y": 0.82,
    "z":   6
  },
  "angles":
  {
    "yaw":   -2,
    "pitch": 1.5,
    "roll":  0
  }
}

---------- Passenger ----------: charlie.com

{
  "validTime": 1674767748003,
  "poseID": "https://charlie.com",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/3",
  "frameSpecification":
  {
```

```
    "authority": "https://ogc.org",
    "id": "PROJCRS[\"GeoPose
Local",+GEOGCS["None)"]+CS[Cartesian,3],+AXIS["x",,ORDER[1],LENGTHUNIT["metre",1]],+AX
IS["y",,ORDER[2],LENGTHUNIT["metre",1]],+AXIS["z",,ORDER[3],LENGTHUNIT["metre",1]]+USA
GE[AREA["+/-1000 m"],BBOX[-1000,-1000,1000,1000],ID["GeoPose",Local]]",
    "parameters": "{\"x": 1234567890.9876,"y": 2345678901.8765, "z": 3456789012.7654}"
  },
  "quaternion":
  {
    "x":0.0174509,"y":0.0130876,"z":-0.0002284,"w":0.9997621
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 4

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/4",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 33.2,
    "y": 0.74,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}


========== Space Train at Local Clock UNIX Time 1674767808003==========

{
  "validTime": 1674767808003,
  "poseID": "https://example.com/nodes/MarsExpress/1",
  "frameSpecification":
  {
    "authority": "https://www.iers.org/",
    "id": "icrf3",
    "parameters": "{\"x": 1234567890.9876,"y": 2345678901.8765, "z": 3456789012.7654}"
  },
  "quaternion":
  {
    "x":0,"y":0,"z":0,"w":1
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 1
```

```
{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/1",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 2.2,
    "y": 0.82,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}

---------- Passenger ----------: Alice

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Passengers/Alice",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/1",
  "position":
  {
    "x": 2.2,
    "y": 0.8,
    "z":   -7
  },
  "angles":
  {
    "yaw":   180,
    "pitch": 1,
    "roll":  0
  }
}

---------- Passenger ----------: Bob

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Passengers/Bob",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/1",
  "position":
  {
    "x": 2,
    "y": 0.8,
    "z":   -6
  },
  "angles":
  {
    "yaw":   180,
```

```
    "pitch": 2,
    "roll":  0
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 2

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/2",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 12.2,
    "y": 0.78,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 3

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/3",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 22.5,
    "y": 0.77,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}

---------- Passenger ----------: Carol

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Passengers/Carol",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/3",
  "position":
  {
    "x": -5,
```

```
    "y": 0.82,
    "z":   6
  },
  "angles":
  {
    "yaw":   -2,
    "pitch": 1.5,
    "roll":  0
  }
}

=-=-=-=-=- Wagon -=-=-=-=-=: 4

{
  "poseID": "https://example.com/nodes/MarsExpress/1/Wagons/4",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1",
  "position":
  {
    "x": 33.2,
    "y": 0.74,
    "z":   -7
  },
  "angles":
  {
    "yaw":   0.2,
    "pitch": 0,
    "roll":  23
  }
}

---------- Passenger ----------: charlie.com

{
  "validTime": 1674767808330,
  "poseID": "https://charlie.com",
  "parentPoseID": "https://example.com/nodes/MarsExpress/1/Wagons/4",
  "frameSpecification":
  {
    "authority": "https://ogc.org",
    "id": "PROJCRS[\"GeoPose
Local",+GEOGCS["None)"]+CS[Cartesian,3],+AXIS["x",,ORDER[1],LENGTHUNIT["metre",1]],+AX
IS["y",,ORDER[2],LENGTHUNIT["metre",1]],+AXIS["z",,ORDER[3],LENGTHUNIT["metre",1]]+USA
GE[AREA["+/-1000 m"],BBOX[-1000,-1000,1000,1000],ID["GeoPose",Local]]",
    "parameters": "{\"x": 1234567890.9876,"y": 2345678901.8765, "z": 3456789012.7654}"
  },
  "quaternion":
  {
    "x":0.0174509,"y":0.0130876,"z":-0.0002284,"w":0.9997621
  }
}
```

```
Enter to exit
```

Copyright and License (MIT)

References

C++: standard and GitHub repo

C# 6 standard and C# 11 features

EPSG:International Association of Oil & Gas Producers geodetic parameter dataset

OGC GeoPose 1.0

IEEE 754-2019 IEEE standard for floating point arithmetic

Java Platform SE 8

JavaScript programming language

Kotlin programming language

NASA SPICE system

The .NET open source ecosystem, including .NET 6 and later.

Open Geospatial Consortium (OGC)

Python programming language

SEDRIS Spatial Reference Model

Structure from Motion (SfM)

Simultaneous Localization and mapping (SLAM)

Swift programming language

TypeScript programming language