Design Pattern for Multi-Modal Coordinate Spaces

Daniel Ruijters[1],
Jeroen Terwisscha van Scheltinga[2], Bart M. ter Haar Romeny[3], Paul Suetens[4]

[1] Philips Medical Systems, X-Ray Predevelopment,
Veenpluis 6, 5680DA Best, the Netherlands, danny.ruijters@philips.com
[2] Philips Medical Systems, Cardio-vascular, XtraVision software,
Veenpluis 6, 5680DA Best, the Netherlands
[3] Technische Universiteit Eindhoven, Biomedical Engineering,
Image Analysis and Interpretation, den Dolech 2, 5600MB Eindhoven, the Netherlands
[4] Katholieke Universiteit Leuven, Medical Image Computing, ESAT/Radiologie,
Herestraat 49, B-3000 Leuven, Belgium

Abstract

Applications for visualizing and/or modelling of complex 3D scenes, such as programs dealing with medical multi-modal 3D data sets, have to cope with a vast number of coordinate spaces. In such applications, treating a datum in the wrong coordinate space is one of the most common reason for bugs. Furthermore, the large number of coordinate transformations lead to code which is difficult to handle and understand. In this article a framework is presented that encapsulates coordinate space transitions, and offers the user a simple and logical entry point to access such transformations. It reduces the chance of 'assuming' a wrong coordinate space to a minimum. The framework is very flexible, in the sense that it allows dynamic composition of scenes, with dynamic transformations between the scene objects. It further allows any type of transformation: affine, non-affine, many-to-one projections, discontinuities, etc.

1 Introduction

There are numerous applications with a large number of geometrical coordinate spaces (also known as coordinate systems). This is the case in e.g. interactive 3D graphics and modelling applications, medical applications dealing with (multi-modal) voxel volumes, algorithms handling complex dynamic mechanical structures, coordinate transformations for astronomical or geodetic purposes (such as GPS), and many others. The relationships between the different coordinate spaces can be static or dynamic, bijective or many-to-one, continues or contain discontinuities, and be affine or non-affine.
When the amount of coordinate spaces is large, the code dealing with coordinate transformations may become complex and prone to errors. To overcome these problems, we developed a software design pattern, to transparently and robustly deal with multiple coordinate spaces. This framework especially aims at severely reducing the chance of making false assumptions, and reducing the complexity of the code.
The framework can be applied for the transformation of data between related spaces of ordered *n*-tuples, such as vector spaces. It might be of interest to point out that these spaces are not necessarily Euclidean. For instance, a voxel space (a vector space representing voxel indices), whereby the voxels are not cubic (which is very common in e.g CT and MR volumes) is *not* Euclidean. Strictly speaking, the design pattern can also be applied to spaces of ordered *n*-tuples that are not vector spaces, i.e. spaces whose elements cannot be linear combined, such as e.g. manifolds. The only criterion is that a mapping exists between the spaces.

2 Related work

An intuitive hierarchical data structure for managing the relations between different coordinate systems is the scene graph. Scene graphs are used in numerous graphics applications, toolkits, modelling and programming languages, such as VRML [1], Open Inventor [2], OpenGL Performer [3], Java 3D [4], Open SG [5], Open Scene Graph [6], nVidia NVSG [7] and many others. Where VRML is only able to describe the scene graph, the others also provide some means to transform points from one coordinate system to another. The transformations are, however, still very much driven by the user of these tools, i.e. it is his responsibility to keep track of which data are in which coordinate space, and whether and how it should be transformed. The framework that is presented here, is not intended to replace these toolkits. It rather is build on top of a scene graph, and thus could be used as an extension to the mentioned toolkits.
Zuiderveld and Viergever [8] describe an Object-Oriented approach aimed at integrated visualization of multiple volumetric datasets, which also deals with coordinate systems. However, they chose to leave the coordinate transformations to the responsibility of the user of their framework. Nadeau [9] presents volume scene graphs, a structure for composing scenes containing volumetric data sets, where the scene graph is used to transform coordinates from world to image space. Other transformations, though, are not directly provided by his framework.
In geospatial applications [10, 11] it often is desirable to express points in different geometry systems, such as geocentric, heliocentric or local coordinate systems. For reasons of efficiency or simplicity it can be desirable to express coordinates in flat earth or spherical earth coordinate systems, and for accuracy ellipsoid or geoid coordinate space may be required. The software paradigm in this paper can be used to easily query data in the desired coordinate system.
There is a significant difference between a point and an extent, as Weisert [12] points out. This difference is particularly of importance when transforming data from one coordinate system to another one (e.g. translations do not affect the values of an extent, but they do affect the coordinates of a point).
The here presented framework can be regarded as a software design pattern for dealing with large number of coordinate spaces. A general overview of design patterns is offered by Gamma et al. (“the gang of four”) [13].

3 Design basics

We define “geometry classes” as a set of object classes, describing basic geometrical entities, such as points, vectors, lines, angles, planes, etc. To identify them easily, we use the prefix “*geo*”. Instances of these classes are generalized under the term “geometry objects”. It is our objective to easily query them in any given coordinate space. A further important class in our framework, is the *3D entity*, which is a node in the scene graph. Any spatial object that can be drawn should be derived from the *3D entity* class. But there can be also abstract *3D entities*, that do not draw anything. A *3D entity* contains a number of geometry objects, to describe its spatial properties.
One of the first observations we made, is the fact that any *3D entity*, which can be found in a scene graph, implicitly defines its own coordinate space. Consider a traditional *3D entity* (e.g. a table), which is located in a parent space (e.g. a room). The object has a translation, which corresponds to the coordinate of the origin of the object expressed in its parent space. Further it can have a rotation, which corresponds to a rotation of its axes with regard to the axes of the parent space, and a scaling. In fact we have just described a rigid transformation between two coordinate spaces.

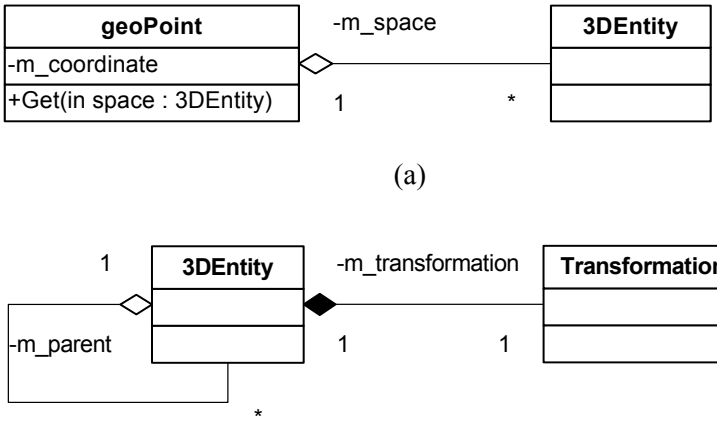

Figure 1: (a) the *geoPoint* class has internal coordinate values, and a reference to its internal coordinate space, (b) every *3DEntity* contains a reference to its parent, and a spatial transformation with regard to its parent.

In our software paradigm, an abstract coordinate space is an instance of a *3D entity*, and therefore every instance of a class, inherited from the *3D entity* class, always defines its own coordinate space.
Further, we established that the position of a point is always defined in a coordinate space. This may be a trivial observation. In an application with many coordinate spaces, though, treating a datum in the wrong coordinate space is one of the most common causes for bugs, and may be difficult to track when coordinate spaces are similar or related. The same considerations are of course true for instances of other geometry classes, e.g. normals, lines, planes, etc. Therefore we provide all geometry classes, with a reference to the coordinate space they are internally defined in (see Figure 1a). In this way it is virtually impossible to 'assume' a wrong coordinate space.
The internal coordinate space of an instance of such a class is defined at construction of the instance, and stays fixed during the lifetime of the instance. This is particularly of importance when the relations between the coordinate spaces are dynamic. Imagine, for instance, a camera, which moves with respect to the depicted scene; the relation between the camera coordinate space and scene coordinate space is then dynamic. It matters whether the coordinates of a point are defined in camera space (e.g. relative to the view port corners), or in scene coordinates (e.g. relative to the position of an object in the scene).

4 Transforming space

The *Transformation* class describes the spatial mapping between an instance of a *3D entity* and its parent in the scene graph (see Figure 1b). This is an abstract class, and specific transformation classes, such as affine transformations, are inherited from this class.
The *Transformation* class possesses virtual functions to transform coordinates from its owner space to the parent of its owner, and vice versa (this approach is similar to the Visitor design pattern [13]). These functions return a boolean to indicate whether the requested transformation could be performed. In this way also many-to-one relations could be implemented; the function corresponding to the one-to-many direction would then always return false. Further, transformations that are only valid for a certain sub-space can use this mechanism, since they would return false for points outside the sub-space. Similar virtual functions are available for transforming all other geometry objects, like vectors, matrices, plane equations, etc.
A rigid transformation can be described by translation and rotation only. In the case of an affine transformation (of which the rigid transformation is a sub class) the virtual

transformation functions can be implemented by multiplying homogeneous coordinates (*x, y, z, w*), representing points or vectors, with the 4*4 transformation matrix (or inverted matrix, for the inverse transformation). A typical implementation for elastic (non-affine) transformations could use spline interpolation, driven by a volumetric mesh.
Since no assumptions are being made about the type of transformation, the various relations in a scene graph might be of a different kind (e.g. affine and non-affine transformations could be found in the same scene graph, to depict for instance underwater scenes).

5 Querying geometry objects

One of the most essential functions that any geometry class has in our framework, is the `Get` function (see Figure 1a). The `Get` function allows to query a geometry object with respect to a given coordinate space. The `Get` function of e.g. the *geoPoint* class takes a reference to a coordinate space as input parameter, and returns the coordinate values of the point with respect to the passed coordinate space.
If the passed reference to a coordinate space equals the internal space of a geometry object, its internal values are simply returned. If they are not equal, the internal values are transformed from the internal coordinate space to the destination one, passed as input parameter. In order to do this, the scene graph is traversed, delivering the path from the internal coordinate space to the destination one. The transformations between the intermediate coordinate spaces in the path are then applied to the geometry object. If a transformation returns false (thus it is not possible to transform the geometry object over that node), or if no path exists between the internal and destination space (i.e. they are not in the same scene graph), an exception is thrown.
Four rigid transformations of a coordinate, as is shown in Figure 2a, take 2.7 μs on a Pentium IV 3.0 GHz machine.

6 Traversing the scene graph

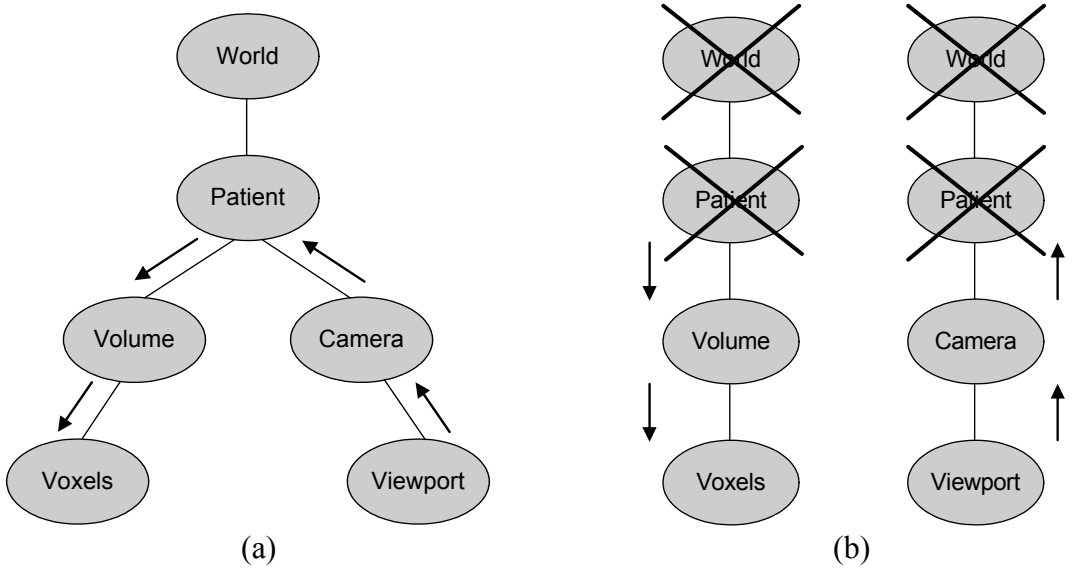

Figure 2: (a) Traversing the scene graph, (b) Building the path that represents the traversal of the scene graph.

When coordinate space that is passed to the `Get` function of geometry object, is different than the internal one, the scene graph has to be traversed. In order to perform this efficiently two arrays with references to the nodes in the scene graph are built. The ascending array starts

with internal coordinate space of the geometry object. Iteratively the parent of the last node in the array is added, until the top node is reached. The descending array starts with the destination coordinate space, and also here parents are added until the top node is reached. Now it is checked whether the last node (the top node) in both arrays is the same. If this is not the case, meaning that the internal and destination coordinate spaces are not in the same scene graph, and an exception is thrown.
After the check, the nodes at the end of both arrays are removed if they are the same. This is repeated until the ends are different, see Figure 2b.
The remaining nodes now form the path that has to be traversed. The data of the geometry object is consequently transformed by the nodes in the ascending array, starting with first node in the array. Then the data is transformed by the nodes in the descending array, calling the inverse transformation functions. This array is parsed starting from the end, see Figure 3. Note that the order of this algorithm is only determined by the height of the scene graph, not by its width.

7 Operator overloading

Another important feature is the fact that we overloaded the operators of the *geoPoint* and *geoVector* classes. Note that two points cannot be added together, but a point and a vector can be added, delivering a new point [12]. By overloading the operators, we can even add points and vectors, which are internally expressed in a different coordinate system.
For instance let us consider the position of an object in the scene, expressed by an instance of the *geoPoint* class, with the world coordinate system as internal space (expressed in e.g. millimeters), and a *geoVector* instance, expressing a mouse movement, with the view port coordinate system as internal space (in pixel coordinates). Suppose we want to translate the object by the mouse movement. The corresponding code could be as simple as `objPos = objPos + mouseMove;` (see Figure 3). Note that the internal coordinate spaces of the variables in this expression is different.
The overloaded `operator+` of the *geoPoint* class will take care that `mouseMove` is transformed to the coordinate space of `objPos`. The `Get` member function of the *geoVector* class will transform the internal values of the `vec` variable from its own internal space to the internal space of the `objPos` variable, and then the two can be added without any problems. This feature leads to very powerful and simple code, as illustrated in the `TranslateObject` function, since the code expresses what you want to achieve conceptually, instead of expressing all kinds of difficult coordinate transformations.

```
void TranslateObject(geoPoint& objPos, const geoVector& mouseMove)
{
    objPos = objPos + mouseMove;
}

geoPoint geoPoint::operator+ (const geoVector& vec) const
{
    return geoPoint(m_coordinates + vec.Get(m_space), m_space);
}
```

Figure 3: The function TranslateObject illustrates the code that a user of the Coordinate Space Design Pattern would write. The operator+ below, shows how the design pattern deals with this code internally. The mouseMove variable is queried in the coordinate space of the objPos variable.

8 Real life examples

8.1 Mouse click on a voxel volume

Take an iso-surface rendered voxel volume, and suppose we want to determine which surface voxel lies under the mouse cursor at a mouse click. In order to do so, a line through the cursor position (viewing ray) has to be intersected with the isosurface. This line is defined by the cursor position and the camera normal, in the case of a parallel projection, and by the cursor position and the camera
focus point in the case of a perspective projection. Using the presented framework, it is no problem to define a line from two points which are constructed in different coordinate systems. After the line has been defined, it can be easily obtained in voxel space, using the `Get` function. Then it should be passed to a 3D variant of Bresenham's algorithm [14], to deliver the intersection point.

8.2 Defining points of interest

Consider an application with two windows next to each other, in order to view two registered multi-modality volumetric data sets. In one view a slice of the reference data set is displayed, while in the other the corresponding interpolated surface through the other data set is shown (non-affine registration). A mouse click marks a point of interest in one data set. The point can be constructed as follows: `geoPoint mousePnt(viewport1.x,viewport1.y,0);` Drawing the corresponding point in the other view is as simple as: `Plot(mousePnt.Get(viewport2));`, assuming that the Plot function is library function that takes pixel coordinates as input.
Note that the `Get` function transforms the `mousePnt` coordinates first from viewport1 (pixels) to world coordinates (millimeters), which is a rigid transformation. Then the coordinates are transformed from world coordinates to the frame of reference of volume2, which is a non-affine transformation, executed by a different transformation class. Finally the coordinates are transformed from the frame of reference to viewport2, which is a rigid transformation again, delivering the pixel coordinates of the point to be plotted. This complete procedure remains hidden for the user of the framework.

8.3 Gantry-tilt CT volumes

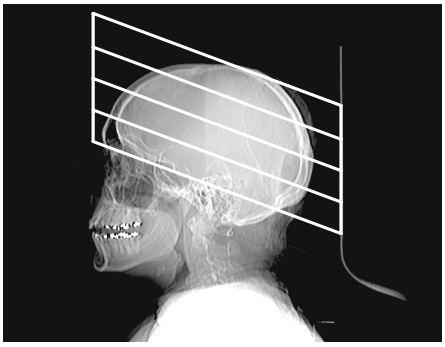

Figure 4: Planning the location of the CT slices, with tilted gantry. The gantry is tilted to avoid radiating the eyes, while capturing a maximum of relevant anatomical data.

CT volumes, which have been acquired with a tilted gantry, produce a voxel space with non-orthogonal axes (see Figure 4). Typically such volumes are resampled on a orthogonal grid for volumetric visualization, leading to loss of image quality. However, it is possible to encapsulate the shearing (skew) that is introduced by the non-orthogonality in an affine transformation (e.g. expressed in a 4*4 matrix). In this way the data can be queried without resampling, using our framework. The fact that the data is stored in a non-orthogonal grid

remains completely hidden for a programmer who accesses it through the coordinate space design pattern, since he can define positions in e.g. patient space (millimeters), and the framework takes care of the transformation to the skewed space.

8.4 Follow camera orientation

Suppose we want one single object in the 3D scene always to be presented with the same side to the camera. This object, however, should be positioned and scaled according to its location in the scene (i.e. moving camera could change only the rotation of the object, but not its translation or scaling). To solve this task, we can define to *geoVector* instances in camera space, representing the x- and y-axis of the camera. For the x-axis this can look like: `geoVector x_camera(1,0,0,cameraSpace);` Now we will rotate the object such that its x-axis will point in the direction of the camera x-axis. To do so we query the camera x-axis in the coordinate space of the object: `x_camera.Get(objectSpace);` The nice thing is that this produces the orientation of the camera x-axis in the object space instantaneously, no matter how many nodes there are between the camera and the object in the scene graph. The vector still has to be normalized, and then the dot product between this vector and the object x-axis (which is simply (1,0,0)) delivers the cosine of the angle that we should rotate. The cross product delivers the rotation axis. The same procedure can be followed to orient the y-axis correctly.

9 Conclusions

In this article we have introduced a generic software solution for a flexible and transparent design pattern for handling multiple coordinate spaces. The proposed framework is especially powerful when the number of coordinate spaces is large and their relations are dynamic, such as is e.g. the case in multimodality medical applications.
The complexity of dealing with multiple coordinate spaces lies in the transformation between the individual spaces. The strength of the proposed framework is the fact that these transformations are maintained at a single spot, and in the rest of the code no awareness of these transformations is needed. The code expresses what you want to achieve conceptually, instead of expressing all kinds of difficult coordinate transformations.
In the case that the actual values of a geometry object are needed with respect to a certain coordinate space, these can be only obtained by explicitly passing the desired coordinate space to the `Get` operation. This severely reduces the chance of 'assuming' a wrong coordinate space, one of the most common causes of bugs in such applications. If a transformation is needed from the internal coordinate space to the requested one, the transformation is performed automatically, and hidden from the user of the function call.
The design pattern has been successfully implemented in two medium and three large scale software projects.

References

[1] International Standards Organization. *The Virtual Reality Modeling Language.* ISO/IEC 14772-1:1997, 1997

[2] D. Wang, I. Herman and G. J. Reynolds. *The Open Inventor Toolkit and the PREMO standard.* Computer Graphics Forum, 16(4): 159–175, 1997

[3] J. Rohlf and J. Helman. *IRIS Performer: a high performance multiprocessing toolkit for real-time 3D graphics.* In Proc. ACM SIGGRAPH, 1994, 381–395

[4] H. A. Sowizral, K. Rushforth and M. Deering. *The Java 3D API specification.* Addison-Wesley, 1998

[5] D. Reiners, G. Voß, and J. Behr. *OpenSG Basic Concepts.* In Proc. OpenSG 2002 Symposium, Darmstadt, 2002.
[6] Open Scene Graph. *http://www.openscenegraph.org/*
[7] nVidia NVSG. *http://developer.nvidia.com/object/nvsg_home.html*
[8] K. J. Zuiderveld and M. A. Viergever. *Multi-Modal Volume Visualization using Object-Oriented Methods.* In Symp. Volume Visualization, ACM SIGGRAPH, 1994, 59–66
[9] D. R. Nadeau. *Volume Scene Graphs.* In Proc. IEEE Symposium on Volume Visualization, 2000, 49–56
[10] A. Sekar and A. H. Lee. *Defining the Universe: Creating a Data Model for a Geospatial Data Repository.* In Proc. SIWs Spring Simulation Interoperability Workshop, 2004
[11] R. M. Toms and P. A. Birkel. *Choosing a Coordinate Framework for Simulations.* In Proc. SISO Fall Simulation Interoperability Workshop, 1999
[12] C. Weisert. *Point-Extent Pattern for Dimensioned Numeric Classes.* ACM SIGPLAN notices, 32(11): 17–20, 1997
[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995
[14] J. E. Bresenham *Algorithm for computer control of a digital plotter.* IBM Systems Journal, 4(1): 25–30, 1965