ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT

Computerized Medical Imaging and Graphics 27 (2003) 255–265

Computerized Medical Imaging and Graphics

www.elsevier.com/locate/compmedimag

Generalized 3D nonlinear transformations for medical imaging: an object-oriented implementation in VTK

David G. Gobbi[a,b,*], Terry M. Peters[a,b]

[a]Imaging Research Laboratories, Robarts Research Institute, London, Canada N6A 5K8
[b]Department of Medical Biophysics, University of Western Ontario, London, Canada

Received 30 October 2000; revised 10 September 2002; accepted 10 September 2002

Abstract

We have contributed an efficient, object-oriented implementation of 3D nonlinear transformations to the Visualization Toolkit that can be applied to a wide variety of data types, including images and polygonal meshes. The transformations are performed via thin-plate splines or via interpolation of a regular lattice of displacement vectors, and are part of a framework that is easily extensible to other nonlinear transformation types. In this paper we demonstrate application of these transformations in medical imaging in general and image-guided surgery in particular, and present a series of performance benchmarks.
© 2002 Elsevier Science Ltd. All rights reserved.

Keywords: Medical imaging; Nonlinear transformation; Thin-plate splines; Visualization; Ultrasound; Surgery

1. Introduction

The visualization toolkit (VTK) [1][1] is an open-source, cross-platform C++ library for 3D data visualization that has a large user base in the medical research community. We have enhanced VTK through the addition of a flexible, object-oriented implementation of nonlinear 3D geometrical transformations. When medical images from different subjects must be compared, or when images of the same subject must be compared where the anatomy has shifted between images, a nonlinear coordinate mapping between the two images spaces is required to provide a one-to-one matching of all the homologous features in the images [2]. These nonlinear mappings are used, for example, in the analysis of the variance of brain morphology across a population [3], in the application of standard anatomical or functional atlases of the brain to particular patients [4–6], and in the warping of 3D pre-operative MR image volumes used for surgical guidance to match the distortion that the brain undergoes during surgery [7–10].

For many projects in our laboratory, we have applied nonlinear geometrical transformations to images using a set of UNIX command-line tools developed at the Montreal Neurological Institute (MNI) by Peter Neelin, David MacDonald and Louis Collins.[2] These tools are flexible, but are designed to be used for batch processing of data rather than for interactive data manipulation. In contrast, VTK is designed specifically for the development of interactive visualization applications but, until now, has not provided the means of applying nonlinear transformations to data.

The geometrical transformations we have added to VTK are implemented as a C++ class hierarchy. All classes in the hierarchy provide an invertible mapping between two coordinate spaces (x', y', z') and (x, y, z). Branches of the hierarchy place additional restrictions on the mapping, e.g. by restricting the transformations to those that can be represented by a 4×4 homogeneous coordinate transformation matrix or to strictly linear coordinate transformations. Arbitrary nonlinear transformations can be represented either as a sampled grid of $(\Delta x, \Delta y, \Delta z)$ vectors, or as a list of homologous points $[(x_n, y_n, z_n) : (x'_n, y'_n, z'_n)]$ where the displacements are interpolated via a thin-plate spline [2] or other radial basis functions.

* Corresponding author. Address: Imaging Research Laboratories, Robarts Research Institute, London N6A 5K8 Canada. Tel.: +1-519-663-5777x34213; fax: +1-519-663-3403.
E-mail address: dgobbi@imaging.robarts.ca (D.G. Gobbi).

[1] Available from http://www.visualizationtoolkit.org/ or http://www.kitware.com/

[2] Available from http://www.bic.mni.mcgill.ca/software/

0895-6111/03/$ - see front matter © 2002 Elsevier Science Ltd. All rights reserved.
doi:10.1016/S0895-6111(02)00091-5

Our implementation also provides the means of combining arbitrary transformations through concatenation and/or inversion operations. The user creates a set of rules that describe the arithmetic relationship between transformations, e.g. 'transformation C is the concatenation of transformation A with the inverse of transformation B,' and a lazy evaluation scheme (meaning that computations are deferred until the results of the computations are required) is used to improve efficiency. This methodology of specifying the relationships between transformations and having the computations deferred until they are necessary is very valuable even for applications that use only linear transformations.

The geometrical transformations can be applied to images or 3D image volumes, to any data represented by a mesh of connected (or unconnected) vertices, or to any scalar field of the form $c = f(x, y, z)$. When the transformations are applied to images, VTK automatically distributes the computations among all processors on the computer. As a result, the time required to perform an image transformation scales inversely with the number of available processors.

1.1. Modifications to VTK

VTK is a very large collection of C++ classes which can be used to perform sophisticated medical, industrial, or scientific data processing and visualization tasks on a variety of data types. VTK is distributed by Kitware, Inc. (469 Clifton Corporate Parkway, Clifton Park, NY 12065 USA) under an open license that permits free inclusion of VTK components into both commercial and non-commercial applications. We have integrated our vtkAbstractTransform class hierarchy into VTK with the permission and support of the original VTK authors, and the C++ classes described in this paper are part of the VTK distribution available from the Kitware web site (http://www.kitware.com/).

1.1.1. Nonlinear transformations and VTK

In VTK 2.4 (released in 1999) geometric transformations in VTK were limited to either affine or perspective operations. This restriction limited the utility of VTK for combining data sets which describe anatomy, both because biological tissues are in general not rigid and because anatomy varies between individuals. As of VTK 3.1 and as a direct result of the work presented here, it is possible to apply nonlinear warp transformations to data where it was previously only possible to apply affine and perspective transformations. The VTK data types to which transformations can be applied are as follows

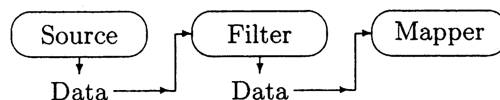
vtkImageData a data set sampled at regular intervals along a 1D, 2D or 3D rectilinear grid.
vtkPolyData a mesh of (x, y, z) vertices that are connected to form a collection of 2D (polygon), 1D (line) and 0D (single vertex) cells.
vtkUnstructuredGrid a mesh of (x, y, z) vertices that are connected to form a combination of 3D, 2D, 1D and 0D cells.
vtkStructuredGrid a mesh of (x, y, z) vertices that are connected so as to be topologically similar to a regular 1D, 2D or 3D rectilinear grid.

For the last three data types, the transformation is simply applied to the (x, y, z) coordinates of each vertex, as well as to surface normals defined at each vertex. For vtkImageData, the data are resampled onto a new grid using nearest-neighbor, trilinear, or tricubic interpolation. The resampling is performed by a C++ class, vtkImageReslice, which we contributed to VTK in 1998 and have enhanced many times since. The vtkImageReslice class takes advantage of the breakdown of transformation types into a hierarchy by providing optimized code paths for perspective, affine, and permutation transformations as well as a general code path for nonlinear warp transformations.

VTK also supports an implicit function type that can be modified by a transformation. The vtkImplicitFunction algebraically defines a scalar field $f(x, y, z)$. Most often the field is used to implicitly describe a surface via the equation $f(x, y, z) = 0$. Implicit functions are used to support a broad variety of visualization tasks that go beyond the scope of this paper.

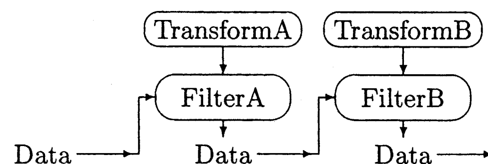
1.1.2. The transformation pipeline

Most visualization systems are described through the analogy of a pipeline, where the data 'flows' through a series of mathematical 'filters.' The VTK data pipeline, as it has existed since the creation of VTK, relies on two very distinct object types: data objects (derived from vtkDataObject) and process objects (derived from vtkProcessObject) which operate on the data. A typical VTK data pipeline has the following structure



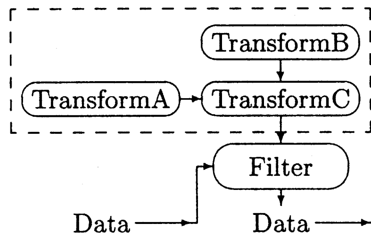
The source, filter, and mapper classes are all derived from vtkProcessObject. The source creates a VTK data object (usually by reading data from a disk file), the filter operates on the data, and the mapper converts the data into a visual representation that can be displayed on the computer screen.

The behavior of certain VTK filter objects, e.g. a vtkTransformPolyDataFilter, can be specified by attaching a VTK transform object to the filter. In the following example, each of FilterA and FilterB apply a transformation to the data.



In practice, one would rarely construct such a pipeline because one of the filters (and hence half the computations) can be eliminated if the transformations are concatenated before the data are filtered. If one of the transformations is modified after the concatenation this approach can lead to problems: the user must keep track of such modifications, and update the concatenated transformation as necessary. The goal of the VTK pipeline mechanism is precisely to keep track of such modifications so that the user does not have to. We perceived as a major shortcoming the fact that the VTK pipeline could not automatically update transformation concatenations.

We have contributed a new pipeline mechanism to VTK which we call the transformation pipeline mechanism. It automatically performs transformation concatenations as necessary, according to how a pipeline has been constructed by the user. The original two-filter pipeline becomes the following.



The new VTK transformation pipeline (inside the dashed border) is distinct from, but connected to the VTK data pipeline. A VTK data pipeline is an explicit set of rules that specify how a data object at any stage of the pipeline is related to data objects at earlier stages of the pipeline, while a VTK transformation pipeline is a set of rules that specify how the *target* transformation can be constructed from other transformations, the *dependencies* of the target. In this example, the transformation pipeline states the following rule: 'TransformC is TransformA concatenated with TransformB.' It is possible to state much more complicated rules, e.g. 'TransformD is TransformA, translated by (10,2,2), concatenated with TransformB, rotated around the x axis by 30°, concatenated with the inverse of TransformC.' The methods used to state these rules in a VTK program are discussed in Section 5.

2. Linear, perspective and general nonlinear transformations

The following subsections provide a brief outline of the mathematical framework that underlies the classes in the vtkAbstractTransform hierarchy.

2.1. Linear and perspective transformations

Linear and perspective transformations are performed using homogeneous coordinates (x_h, y_h, z_h, w_h) according to the familiar equation

$$\begin{pmatrix} x'_h \\ y'_h \\ z'_h \\ w'_h \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x_h \\ y_h \\ z_h \\ w_h \end{pmatrix}, \tag{1}$$

where the homogeneous coordinates are related to the (x, y, z) coordinates through the following expressions:

$$x = x_h/w_h; \qquad y = y_h/w_h; \qquad z = z_h/w_h. \tag{2}$$

For linear transformations, the bottom row of the matrix is (0001) and $w_h = 1$.

For linear and perspective transformations, we perform concatenation via matrix multiplication and inversion via matrix inversion.

2.2. General nonlinear transformations

In general, with the exception of perspective transformations, nonlinear transformations cannot be expressed in terms of matrix multiplication. Instead, the transformed point (x', y', z') is related to the original point (x, y, z) via three nonlinear functions

$$\begin{aligned} x' &= f_x(x, y, z) \\ y' &= f_y(x, y, z) \\ z' &= f_z(x, y, z) \end{aligned} \tag{3}$$

One of the first difficulties with nonlinear transformations is that there is often no analytic solution for their inverse. However, any transformation f that provides one-to-one correspondence between two coordinate spaces does have an inverse, which can be computed numerically if the transformation is smooth and well-behaved.

We use Newton's method to iteratively calculate the inverse transformation of a point (x', y', z') given an initial guess (x_0, y_0, z_0) :

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \begin{pmatrix} 2x' - f_x(x', y', z') \\ 2y' - f_y(x', y', z') \\ 2z' - f_z(x', y', z') \end{pmatrix} \tag{4}$$

$$\begin{pmatrix} x_{(n+1)} \\ y_{(n+1)} \\ z_{(n+1)} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} - J^{-1} \begin{pmatrix} f_x(x_n, y_n, z_n) - x' \\ f_y(x_n, y_n, z_n) - y' \\ f_z(x_n, y_n, z_n) - z' \end{pmatrix} \tag{5}$$

where J is the Jacobian matrix for the transformation

$$J = \begin{pmatrix} \partial f_x/\partial x & \partial f_x/\partial y & \partial f_x/\partial z \\ \partial f_y/\partial x & \partial f_y/\partial y & \partial f_y/\partial z \\ \partial f_z/\partial x & \partial f_z/\partial y & \partial f_z/\partial z \end{pmatrix}. \tag{6}$$

Note that the Jacobian must be re-evaluated at (x_n, y_n, z_n) for each step n of the method. We have implemented a modified version of Newton's method that converges more rapidly and is more stable than the traditional Newton's method: If

the calculated distance d_n between (x', y', z') and the forward transformation of (x_n, y_n, z_n)

$$d_n = [(f_x(x_n, y_n, z_n) - x')^2 + \cdots]^{1/2} \tag{7}$$

increases after any step in the iteration, then a quadratic approximation is used to estimate the point on the line segment between $(x_{(n-1)}, y_{(n-1)}, z_{(n-1)})$ and (x_n, y_n, z_n) that minimizes d. This point is then used for the next step in the iteration. This common tactic is described in Numerical Recipes [11] and other texts on numerical analysis.

3. Transformation of vectors and normals

The transformation of a tangent vector is accomplished through multiplication of the vector (v_x, v_y, v_z) by the Jacobian J

$$\begin{pmatrix} v'_x \\ v'_y \\ v'_z \end{pmatrix} = J \begin{pmatrix} v_x \\ v_y \\ v_x \end{pmatrix}. \tag{8}$$

Normals are often expressed in row format, and are transformed using the inverse of the Jacobian

$$(n'_x \; n'_y \; n'_z) = (n_x \; n_y \; n_z) J^{-1}. \tag{9}$$

The transformed normals must also be converted to unit vectors before they are used in surface lighting calculations, as the transformation does not in general preserve unit length.

The Jacobian can also be used in the transformation of gradient vectors (though the use of Eq. (9) without the normalization step) and also in the transformation of tensors. Such operations are currently not implemented in our software and are not discussed here.

When a perspective transformation is applied via a homogeneous transformation matrix M, we use the well-known method of defining a 'homogeneous component' for the normals in order to avoid inverting the Jacobian at each vertex

$$n_w = -xn_x - yn_y - zn_z \tag{10}$$

$$(n'_x \; n'_y \; n'_z \; n'_w) = (n_x \; n_y \; n_z \; n_w) M^{-1}. \tag{11}$$

Once the transformation is completed, (n'_x, n'_y, n'_z) is re-normalized to unit length and n'_w is discarded.

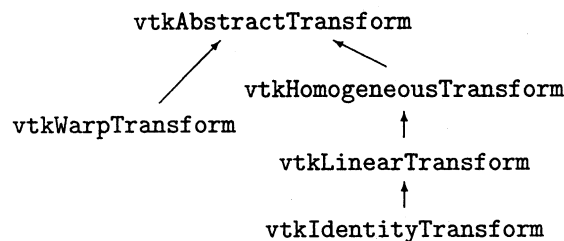
For linear transformations, the Jacobian is invariant with position and need not be recomputed at each vertex for the transformation of either surface normals or tangent vectors.

4. The VTK transform hierarchy

The following subsections describe the interfaces to the individual classes in the vtkAbstractTransform hierarchy. The implementations of the classes are described where they go beyond the mathematical framework discussed in Section 3.

4.1. The vtkAbstractTransform

The vtkAbstractTransform provides an interface for an arbitrary geometrical transformation. The geometrical transformation represented by a vtkAbstractTransform must provide a one-to-one mapping between two $\mathbf{R}^3$ coordinate spaces, but no other restrictions are applied. A skeleton of the vtkAbstractTransform hierarchy is as follows



The vtkHomogeneousTransform and vtkLinearTransform are implemented with 4×4 homogeneous transformation matrices. The vtkLinearTransformation only uses matrices that define linear transformations, i.e. the bottom row of the matrix is (0 0 0 1). The vtkIdentityTransform uses only the identity matrix. The vtkWarpTransform, in contrast, is a base class for transformations that cannot be represented by a 4×4 matrix.

4.1.1. High-level methods

The vtkAbstractTransform class provides methods to apply a transformation to a single point, or to a list of points stored in a vtkPoints object. Transformation of normals and tangent vectors is also supported, but the user must also supply the point at which each normal or vector is defined (refer to Section 3 for the rationale). This restriction is relaxed in the vtkLinearTransform, but is required for all nonlinear transformations.

Finally, the Inverse( ) method inverts the transformation. Because each transformation provides a one-to-one mapping between coordinate systems, every transformation is guaranteed to have an inverse.

```
vtkAbstractTransform: public vtkObject {
public:
  void TransformPoint(double in[3],
    double out[3]);
  virtual void TransformPoints(
    vtkPoints *in,
    vtkPoints *out);
  virtual void TransformPointsNormals-
  Vectors(
    vtkPoints *ip, *op,
    vtkNormals *in, *on,
    vtkVectors *iv, *ov);
  virtual void Inverse();
  ...
};
```

4.1.2. Low-level methods

Three of the methods that are part of the internal implementation of the vtkAbstractTransform are provided as public methods

```
class vtkAbstractTransform:public vtk-
Object {
public:
  ...
  void Update( );
  virtual void InternalTransformPoint(
    double in[3],
    double out[3]);
  virtual void InternalTransformDeriva-
  tive(
    double in[3],
    double out[3],
    double J[3][3]);
protected:
  virtual void InternalUpdate( );

  vtkTimeStamp UpdateTime;
  vtkMutexLock UpdateMutex;
};
```

Calling the Update( ) method causes the transformation to perform any preliminary calculations that it does not have to repeat for each point transformation. For example, a class derived from vtkLinearTransformation will build its 4×4 matrix during the update phase.

The UpdateTime is a timestamp which specifies the last time Update( ) was called. Every VTK object has a timestamp, and the Update( ) method uses the UpdateTime to determine whether or not any changes have occurred to the transformation object that would require computations to be re-performed. The UpdateMutex provides the means of synchronizing different execution threads within VTK to ensure that multiple threads do not execute any state-changing computations simultaneously, which would corrupt the internal data of the transformation object.

The derived classes must define an InternalUpdate( ) method if they require any class-specific update. The derived classes do not have to lock the UpdateMutex or check the UpdateTime because that functionality is provided by the Update( ) method of the base class.

The InternalTransformPoint( ) method performs the transformation without first updating the transformation. It is assumed that the caller of this method will have called Update( ) to ensure that the transformation is up-to-date. The InternalTransformDerivative( ) method performs the transformation, and also returns the Jacobian of the transformation. The Jacobian is used to perform transformations on normals and vectors as described in Section 3, and can also be used to perform transformations on tensors.

4.2. The vtkHomogeneousTransform

The vtkHomogeneousTransform is implemented through a 4×4 matrix that is stored in a vtkMatrix4x4 object. The GetMatrix( ) method provides direct access to the matrix itself. This class is used to represent perspective transformations.

```
class vtkHomogeneousTransform:
  public vtkAbstractTransform {
public:
  vtkMatrix4x4 *GetMatrix( );

protected:
  vtkMatrix4x4 *Matrix;
};
```

4.3. The vtkLinearTransform

The vast majority of geometrical transformations used in computer graphics are linear operations. Because the bottom row of the 4×4 matrix can be ignored and because division by the homogeneous coordinate is not required, linear transformations require at least 25% fewer computations than perspective transformations.

In addition, with linear transformations it is possible to transform normals and vectors without specifying the vertices at which those normals and vectors are defined. Class methods are provided to support this.

4.4. The vtkIdentityTransform

The identity transformation is provided primarily for completeness. The inverse of the identity is the identity, and the Jacobian is a 3×3 identity matrix. It is implemented such that the input points are directly copied to the output points, and no matrix multiplication occurs.

4.5. The vtkWarpTransform

For most nonlinear transformations (excluding perspective transformations) the inverse transformation cannot be analytically computed. Instead, the inverse transformation is computed using Newton's method as described in Section 2.2.

To support this behavior, an InverseFlag is supplied which specifies whether the InternalTransformPoint( ) method results in the computation of the forward transformation or the iterative inverse transformation. The Inverse( ) method simply flips this flag. The derived classes must provide both ForwardTransformPoint( ) and ForwardTransformDerivative( ) methods, where the latter must compute the Jacobian as per Eq. (6). It is not necessary for the derived class to provide InverseTransformPoint( ) or InverseTransformDerivative( ) methods, because these are computed from the forward transformation via Newton's method.

If there is a more efficient means of inversion than using Newton's method for a particular transformation, then the InverseTransformPoint( ) and InverseTransformDerivative( ) can be implemented in the derived class in order to take advantage of this.

```
class vtkWarpTransform:
  public vtkAbstractTransform {
protected:
  void ForwardTransformPoint(
    double in[3], double out[3]);
  void ForwardTransformDerivative(
    double in[3], double out[3],
    double J[3][3]);

  void InverseTransformPoint(…);
  void InverseTransformDerivative(…);

  int InverseFlag;
};
```

4.6. The vtkThinPlateSplineTransform

There are two common representations of warp transformations. The first is a regular three-dimensional lattice of displacement vectors. The second, which is supported by the vtkThinPlateSplineTransform class, is an unstructured set of landmark points in one coordinate space and the set of homologous points in another coordinate space (or, equivalently, one set of landmarks and one set of displacements).

For landmark-based warp transformations, interpolation of the transformation for points that do not lie on a landmark point is achieved using radial basis functions. The thin plate spline [2] is one of the most popular radial basis functions, which is why we chose to name the class 'vtkThinPlateSplineTransform' even though any arbitrary basis function can be used, given a pointer to one C function that evaluates the radial basis function and a pointer to a second which evaluates its first derivative.

```
class vtkThinPlateSplinesTransform:
  public vtkWarpTransform {
public:
  void SetSourceLandmarks(vtkPoints *);
  void SetTargetLandmarks(vtkPoints *);

  void SetBasisToR();
  void SetBasisToR2LogR();
  void SetBasisFunction(
    double (*U)(double r));
  void SetBasisDerivative(
    double(*dU)(double r, double &dU));
};
```

This class includes some heuristics to ensure that if all of the points lie in the XY plane, a 2D thin-plate spline will be performed. Also, if fewer than three landmarks are provided (the minimum number for a thin-plate spline) either a simple translation (one landmark) or similarity transformation (two landmarks) will be applied in place of the thin-plate spline.

4.7. The vtkGridTransform

A grid transformation is represented by a regular three-dimensional grid of displacement vectors. Points that do not lie directly on the grid are transformed by interpolating the displacement vectors using either nearest-neighbor, trilinear, or tricubic interpolation.

Because the grid will often contain 8 or 16 bit integer data instead of floating point, scale and shift variables can be set that specify how the integer displacement vectors are to be converted into real-value displacement vectors.

```
class vtkGridTransform:
  public vtkWarpTransform {
public:
  void SetDisplacementGrid(vtkImage-
  Data *G);
  void  SetDisplacementScale(double
  scale);
  void  SetDisplacementShift(double
  shift);

  void SetInterpolationMode(int mode);
};
```

The grid transformation is much more efficient than the thin-plate spline transformation, particularly when many landmarks are used. Because of this, it is worthwhile to convert a thin-plate spline transformation into a grid if the transformation is to be applied to a large data set. We have provided a special class, vtkTransformToGrid, which converts an arbitrary transformation into a displacement grid that can be used by vtkGridTransform. The grid is only defined over a finite rectangular volume of coordinate space, which must encompass the volume occupied by the data set to be transformed. Choosing an appropriate volume, as well as an appropriate sample spacing for the grid, is the user's responsibility. As the volume increases or the sample spacing decreases, the computational cost of creating the grid increases. Once the grid has been created, it can be applied to multiple data sets.

5. Transformation pipeline implementation

The transformation class hierarchy supports execute-on-demand pipeline behavior as introduced in Section 1.1.2. The mechanism for this pipeline is provided by three concrete classes, one for each of the primary transformation types (general nonlinear, perspective, and linear).

5.1. The new vtkTransform

In VTK version 2.4 and earlier, the vtkTransform was the only class available in VTK to build a transformation and only linear transformations were fully supported. Because this class existed prior to our additions to VTK as described in this paper, we were required to maintain strict backwards compatibility of this class with regards to both its interface and behavior. The following basic interface to the vtkTransform is unchanged from VTK 2.4:

```
class vtkTransform: vtkLinearTransform
{
public:
  void Translate(double x, double y,
  double z);
  void RotateWXYZ(double angle,
    double x, double y, double z);
  void Scale(double x, double y, double z);
  void Concatenate(vtkMatrix4×4 *matrix);

  void PreMultiply();
  void PostMultiply();

  void Identity();
  void Inverse();
};
```

The Translate( ), RotateWXYZ( ), Scale( ), and Concatenate( ) methods allow modification of the transformation. The PreMultiply( ) and PostMultiply( ) methods specify whether the translation, rotation, etc. should be applied to points before or after the transformation already represented by the vtkTransform. OpenGL, for example, follows the PreMultiply convention. This convention is also the default in VTK.

The new vtkTransform features a pipeline mechanism which makes it possible to specify dependency rules that dictate how the *target* transformation is computed from its *dependency* transformations. For example, one can specify that the target is the inverse of another transformation, or that the target is the concatenation of two others. Meaningless rules, such as specifying that a transform is concatenated with itself, are automatically rejected.

A dependency rule is most often stated through the SetInput( ) method. For example if we have transformations `t1` and `t2`, and call `t2` → `SetInput(t1)` followed by `t2` → `Translate(10,0,0)`, then `t2` will always be the same as `t1` but with the extra translation. The pipeline mechanism is supported through code in the InternalUpdate( ) method that ensures that if the input transformation is modified, the new transformation will automatically update itself accordingly. The PreMultiply and PostMultiply methods can be used to specify whether each new transformation should be applied before or after the input transformation.

Dependency rules can also be stated through a second version of the Concatenate( ) method that accepts a vtkLinearTransform instead of a vtkMatrix4 × 4. All of the concatenated transformations and matrices, including those created via Translate( ), Scale( ) etc. are stored in a helper class called the vtkTransformConcatenation. The InternalUpdate( ) method is responsible for concatenating all of the matrices together.

If Inverse( ) is called on a transformation that has an input, then the inversion is pipelined in a similar manner to the concatenation.

5.2. The vtkPerspectiveTransform

The vtkTransform class only supports pipelining of transformations derived from the vtkLinearTransform base class. Pipelining of perspective transformations is provided through the vtkPerspectiveTransform class, which supports all of the vtkTransform methods described above as well as methods for creating special-purpose perspective transformations.

The vtkPerspectiveTransform and vtkTransform classes were not amalgamated into a single class, even though they provide similar functionality, in order to ensure strict typing between linear and perspective transformations. Very little code duplication was necessary because nearly all of the shared functionality is implemented in the vtkTransformConcatenation helper class.

5.3. The vtkGeneralTransform

The vtkGeneralTransform also provides the same functionality as vtkTransform, except that it allows concatenation of any class derived from vtkAbstractTransform. Because not all transformations can be represented with 4×4 matrices, the InternalTransformPoint( ) method must pass the point through each one of the concatenated transformations in turn. As a result, this class is not nearly as efficient as either vtkTransform or vtkPerspectiveTransform.

6. Applications of nonlinear transformations

The image visualization and interactive image-guided surgery software applications developed in our laboratory utilize a set of very high-level visualization modules that are written in the object-oriented scripting language Python.[3] These modules are implemented using VTK, via the scripting-language interface that comes with VTK, and form a solid rapid application development framework that is specifically geared towards medical image visualization [12]. We have placed this framework under an open-source license and made it available from http://www.atamai.com.

[3] Available from http://www.python.org/

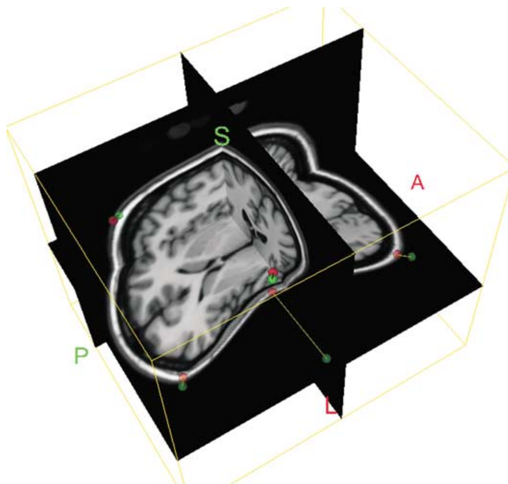


Fig. 1. A thin-plate spline transformation applied to an MR image volume. Five of the eight homologous landmark pairs which define the transformation are visible.

One such module provides a user interface for interactively modifying a thin-plate spline transformation and immediately viewing the result of applying the transformation to a 3D image. Landmark points for the thin-plate spline are placed with the mouse and appear as pairs of semi-transparent spheres, green for the source coordinate system and red for the target coordinate system. The mouse is also used to interactively shift the positions of the landmarks, causing the resulting transformation to change. For a typical $256 \times 256 \times 120$ voxel data set, approximately 0.22 s is required to render a slice through the warped data after a landmark has been moved on a Pentium III 2×933 MHz PC. Usually three orthogonal slices through the data are rendered simultaneously, requiring a total of 0.67 s. Fig. 1 provides an example of a thin-plate spline transformation applied to an MR image volume.

A slice through the warped volume is rendered as follows. First, a grid transform is created which covers the same area and lies at the same position as the image slice but has a lattice spacing that is four times the voxel spacing for the slice. Then, for each node in the grid lattice, the thin-plate spline is used to compute the inverse deformation vector for that node, i.e. to find the (x, y, z) coordinate in the original image which, after it is subject to the thin-plate spline transformation, will be located at the grid node. Once the inverse deformation vectors have been computed for each grid node on the slice, the inverse deformation vectors for each voxel in the slice are computed through cubic interpolation of the vectors at the surrounding nodes. Finally, the vector for each voxel is used to locate the (x, y, z) coordinate in the original image from which the grey value of the voxel is interpolated.

The reason that the thin-plate spline is not used to directly compute the inverse deformation vector at each voxel is that the cost of a thin-plate spline transformation increases linearly with the number of landmarks in the spline. Because the thin-plate spline is evaluated at only 1/16 of the voxel positions (a factor of 1/4 in each of two dimensions), the total rendering speed is increased by a factor of up to 16 when the number of landmark points in the spline is large.

In Fig. 2, a nonlinear transformation has been applied to polygonal data and volume rendered data as well as to the slice views of the MR image volume. The transformation of a volume-rendered image requires that the transformation be applied to the entire image volume, and the result of changing the transformation cannot be viewed interactively. For polygonal data, however, less than a second is required for the transformation unless the data consists of over 100,000 vertices. For both polygonal data and volume-rendered data, the thin-plate spline is converted into a displacement grid that is interpolated to provide the final transformation.

6.1. Performance for image warping

To evaluate the performance of our implementation, we applied different kinds of transformations to an image with 1 mm spacing and dimensions of $181 \times 217 \times 181$ (7.1

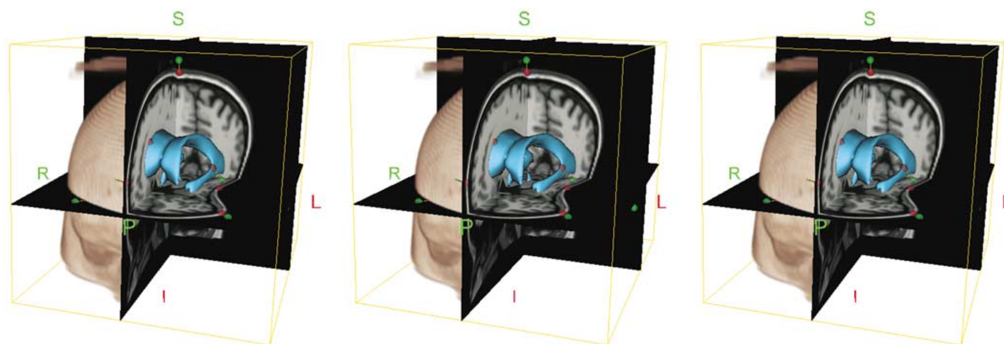


Fig. 2. A thin-plate spline transformation applied to an MR image volume, where a the portion of the image on the left is volume-rendered. The transformation has also been applied to a polygonal surface that defines the surface of the ventricles. Note that this includes the application of the transformation to all surface normals that are stored with the polygonal data. The two images on the right form a stereo pair for cross-eyed viewing, while the two on the left form a stereo pair for parallel-eyed viewing.

million voxels). The transformations applied were as follows.

- linear transformation: a 3×4 matrix
- grid transformation: displacement vectors spaced 4 mm apart on a regular cubic lattice (displacement vectors for coordinates in between lattice nodes are calculated by cubic interpolation)
- thin-plate spline: 27 homologous landmark points, with the 3D thin-plate radial basis function $U(r) = r$.
- inverse grid transformation: the inverse grid transformation, evaluated via Newton's method
- inverse thin-plate spline: the inverse thin-plate spline transformation, evaluated via Newton's method.

Application of the transformation to the image involves, for each voxel, one transformation calculation and one image interpolation calculation. The image interpolation modes available are nearest-neighbor, trilinear, and tricubic.

The performance metric we use in our benchmarks is the the number of voxel transformation/interpolation operations performed per millisecond. For a 2-CPU 933 MHz Intel Pentium III computer with the Linux-2.2 operating system, where our code was compiled with the GNU C++ compiler gcc-2.95, the performance for different transformation types and interpolation modes are given in Table 1. Note the execution time required for a thin-plate spline is proportional to $N + 4$ where N is the number of point-pairs used in the spline, and in our benchmarks we have only tested $N = 27$.

From the data in the table, it can be determined that image interpolation calculations require approximately 10–20% of the total execution time for nonlinear transformations. Because the extra cost of using high-order interpolation is so small, we generally use cubic interpolation for warping images. The cost of applying an inverse transformation via Newton's method is five times as high as for applying a forward transformation, therefore any image warping protocols (i.e. for image registration) should be formulated such that the forward transformation, rather than the inverse, is evaluated unless the latter is explicitly required.

To compare the performance of our implementation on different computer platforms, we applied the grid transformation described above with cubic image interpolation on each of the systems in Table 2. The results, in voxel transformation/interpolation operations per millisecond, are given in the final column. For all systems, the '-o2' level of compiler optimization was used and the compiler flags were set to optimize for the specific CPU architecture of the computer. The development and testing of our code was done primarily under Linux and IRIX, therefore the implementation is expected to favor those two systems. For our code, the MIPS CPUs are able to offer similar performance to PowerPC or Pentium CPUs running at double the clock speed.

The final set of benchmarks were performed to test how efficiently the image transformations could be performed in parallel on symmetric multi-processor (SMP) computer platforms. All of the image filters in VTK automatically run parallel execution threads on each CPU in the computer, and VTK also provides an option to specifically set the number of execution threads that are used. The number of voxel transformations per millisecond will ideally be proportional to the number of CPUs, however, this is only an approximation because the CPUs must share system memory. Nonlinear transformations require many mathematical operations to be performed for each voxel read in from and written out to system memory, therefore we expect the performance of our code to scale nearly proportional to the number of CPUs. The performance scaling factors for three different platforms are given in Table 3.

It is also possible to spread the execution of an image filter in VTK over several computers on a single network in order to reduce the overall execution time, and we intend to investigate the utility of this feature for image warping in the near future.

Table 1
Transformation rate (voxels ms^{-1}) of various transformations with different image interpolation modes

	Transformation type				
	Linear	Grid	TPS	Grid^{-1}	TPS^{-1}
Nearest	7008	529	311	108	75
Linear	3980	512	306	107	73
Cubic	1397	422	276	101	70

Table 2
Performance of image warping on different computer platforms

	CPUs	Speed (MHz)	Architecture	OS	Compiler	Rate (ms^{-1})
(a)	2	933	Intel Pentium III	Linux 2.2	GNU gcc 2.95	101
(b)	2	400	MIPS R12000	IRIX 6.5	MIPSpro 7.2.1	79
(c)	4	225	MIPS R10000	IRIX 6.5	MIPSpro 7.2.1	70
(d)	1	500	IBM PowerPC G4	Mac OSX	GNU gcc 2.95	24

Table 3
Relative performance vs. number of CPUs

CPUs	Pentium III (a)	R12000 (b)	R10000 (c)
1	1.00	1.00	1.00
2	1.96	1.84	1.86
3			2.45
4			3.16

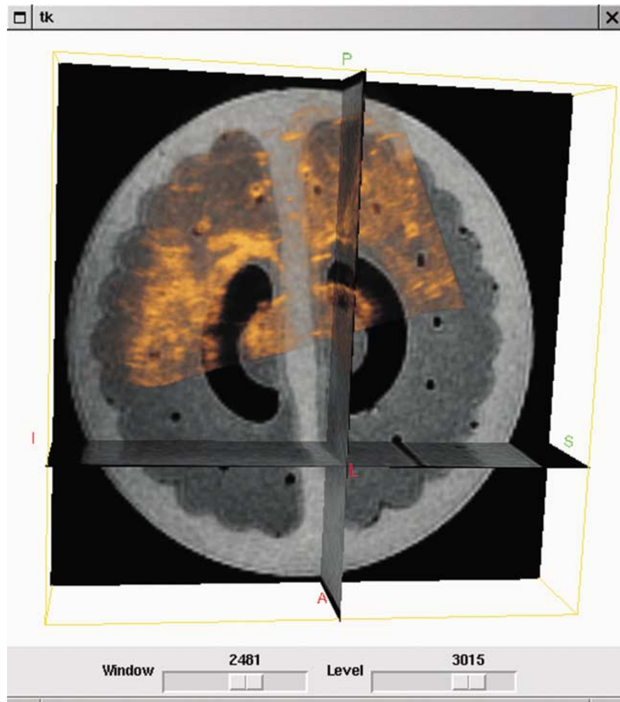


Fig. 3. A 3D freehand ultrasound scan of a phantom superimposed over an MR image volume of the same phantom. The images are not registered.

6.2. Image registration

Our particular interest is the nonlinear transformation of pre-operative MR image volumes to match freehand 3D ultrasound scans that are acquired during neurosurgery. The ultrasound images accurately reflect the positions of structures in the brain, whereas the original MR images will be in error if the patient's brain shifts within the cranium. For craniotomy procedures, this shift is on the order of 10 mm [13]. The use of intraoperative ultrasound in this manner was proposed by Bucholz [14,15] and has been demonstrated clinically by our group [7].

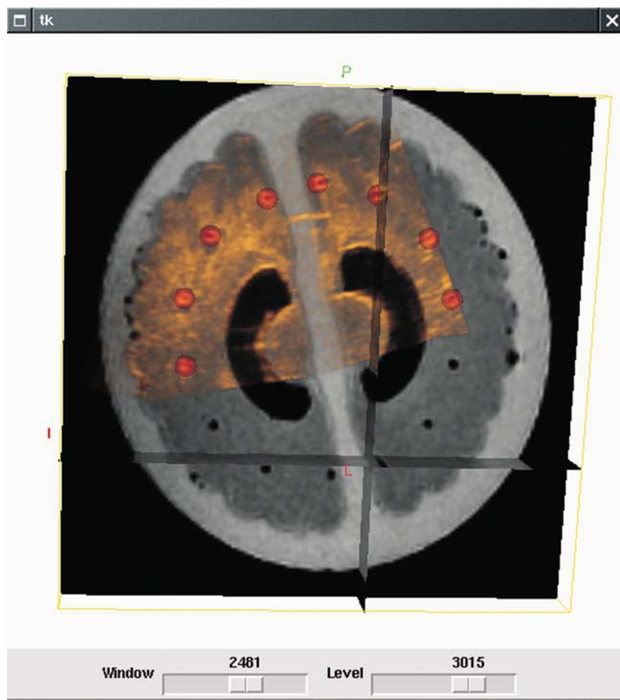


Fig. 4. The images have been interactively registered via a thin-plate spline transformation defined by homologous points pairs in the images. Only one point in each pair is visible, the others are hidden behind the image slice that is being displayed. Note that there is still slight misregistration of the left hemispherical cavity (these cavities were designed to mimic the ventricles in the brain).

As a brief example of how nonlinear registration of the images could be achieved in the operating room, we obtained MRI and 3D freehand ultrasound scans of a PVA-cryogel phantom [9]. Fig. 3 displays the images in their original, unregistered state: the MR is displayed as greyscale and the ultrasound is displayed as orange. Then, we interactively placed landmarks on the images to define a thin-plate spline transformation between the two image coordinate spaces. The result of applying this thin-plate spline to the MR image volume is displayed in Fig. 4.

7. Conclusions

We have developed a hierarchy of C++ classes for computation of linear and nonlinear geometrical transformations on a variety of data types. These have become a part of VTK, the source code for which is available for download from the VTK website.[1] The result is a flexible and scalable framework for geometrical transformations of medical and other data sets, included within a package designed for interactive visualization and data manipulation and capable of taking full advantage of multi-CPU platforms. This combination is particularly valuable for the development of image-guided surgery systems or surgical simulation systems, where interactivity and tight integration of computational modules into the end-user application are strict requirements.

The nonlinear transformation classes are undergoing continual improvement by our group and by others. At the level of the transformation classes themselves, we are experimenting with optimizations to increase the speed of applying certain types of grid and thin-plate spline transformations to images by a factor of at least three via partial decomposition of the transformation along the x, y, and z directions. At the application level, we are developing new tools for automatic nonlinear image registration that utilize these transformation classes.

Acknowledgements

We have received assistance from a large number of people in the completion of this project. Foremost, the integration of our C++ classes into VTK was assisted by discussions with Ken Martin, Lisa Avila and Will Schroeder of Kitware, Inc., as well as Bill Lorensen of General Electric Corporate Research and Development. A portion of the source code in the vtkThinPlateSplineTransform class was

originally contributed to VTK by Tim Hutton, who also provided moral support for our efforts during the early stages of this project. Yves Starreveld receives special thanks for his contributions to the development and design of rapid-application development framework that we use to develop our visualization applications. This project was funded by the Canadian Institutes for Health Research (grant GR14973) and by the Institute for Robotics and Intelligent Systems (Canadian National Centre of Excellence). Graduate student funding for David Gobbi was provided by scholarships from the Ontario Ministry of Education and by the University of Western Ontario.

Appendix A

Class hierarchy diagrams

References

[1] Schroeder W, Martin KW, Lorensen W. The visualization toolkit, 2nd ed. Toronto: Prentice Hall; 1998.

[2] Bookstein FL. Shape and the information in medical image: a decade of the morphometric synthesis. Comput Vision Image Understanding 1997;66:97–118.

[3] Thompson PM, MacDonald D, Mega MS, Holmes CJ, Evans AC, Toga AW. Detection and mapping of abnormal brain structure with a probabilistic atlas of cortical surfaces. J Comput Assisted Tomography 1997;21:567–81.

[4] St-Jean P, Sadikot AF, Collins L, Clonda D, Kasrai R, Evans AC, Peters TM. Automated atlas integration and interactive three-dimensional visualization tools for planning and guidance in functional neurosurgery. IEEE Trans Med Imaging 1998;17:672–80.

[5] Finnis K, Starreveld YP, Parrent AG, Peters TM. A 3-dimensional database of deep brain functional anatomy and its application to image-guided neurosurgery. Medical image computing and computer-assisted intervention—MICCAI, Pittsburg October 11–13. Lecture Notes Comput Sci 2000;1935:1–8.

[6] Finnis K, Starreveld YP, Parrent AG, Peters TM. Three dimensional database of subcortical electrophysiology for image guided stereotactic functional neurosurgery. IEEE Trans Med Imaging 2002;21(11).

[7] Comeau RM, Sadikot AF, Fenster A, Peters TM. Intraoperative ultrasound for guidance and tissue shift correction in image-guided surgery. Med Phys 2000;27:787–800.

[8] Gobbi DG, Lee BKH, Peters TM. Correlation of pre-operative MRI and intra-operative 3D ultrasound to measure brain tissue shift. SPIE medical imaging, San Diego, February 18–20. Proc SPIE 2001;4319: 264–71.

[9] Gobbi DG, Comeau RM, Peters TM. Ultrasound/MRI overlay with image warping for neurosurgery. Medical image computing and computer-assisted intervention—MICCAI, Pittsburg October 11–13. Lecture Notes Comput Sci 2000;1935:106–14.

[10] Jödicke A, Deinsberger W, Erbe H, Kriete A, Böker D-K. Intraoperative three-dimensional ultrasonography: an approach to register brain shift using multidimensional image processing. Minimally Invasive Neurosurg 1998;41:13–19.

[11] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical recipes in C, 2nd ed. Port Chester, NY: Cambridge University Press; 1993.

[12] Starreveld YP, Gobbi DG, Finnis K, Peters TM. Software components for medical image visualization and surgical planning. SPIE medical imaging, San Diego, February 18–20. Proc SPIE 2001;4319:546–56.

[13] Roberts DW, Hartov A, Kennedy FE, Miga MI, Paulsen KD. Intraoperative brain shift and deformation: a quantitative analysis of cortical displacement in 28 cases. Neurosurgery 1998;43: 749–60.

[14] Bucholz RD, Yeh D, Trobaugh J, McDurmont LL, Sturm CD, Baumann C, Jaimie MH. The correction of stereotactic inaccuraccy caused by brain shift using an intraoperative ultrasound device. In: Troccaz J, Grimson E, Mösges R, editors. CVRMed-MRCAS '97: First Joint Conference Computer Vision, Virtual Reality and Robotics in Medicine and Medical Robotics and Computer-Assisted Surgery, Berlin: Springer; 1997. p. 459–66.

[15] Trobaugh WT, Richard WD, Smith KR, Bucholz RD. Frameless stereotactic ultrasonography: methods and applications. Comput Med Imaging Graph 1994;18:235–46.

David G. Gobbi received his MSc from Carleton University in Ottawa and is currently working on his PhD at the University of Western Ontario in London, Canada. His research interests include image processing, 3D graphics and visualization.

Terry M. Peters received his PhD in Electrical Engineering from the University of Canturbury, New Zealand, in 1973. He is currently a scientist with the Robarts Research Institute in London, Canada, and is a professor in the departments of Radiology & Nuclear Medicine and Medical Biophysics at the University of Western Ontario. His principal research area is image-guided surgery and therapy.