TR－A－0170

A software library of C++ class objects for biological structure modeling.

Reiner Wilhelms

1993. 3.23

ATR視聴覚機構研究所

〒619-02 京都府相楽郡精華町光台2-2 ☎07749-5-1411

ATR Auditory and Visual Perception Research Laboratories

2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan
Telephone: +81-7749-5-1411
Facsimile: +81-7749-5-1408

Contents

A software library of C++ class objects for biological structure modeling.

Reiner Wilhelms, ATR Hearing Dept.

September 29, 1992

1 Introduction

Finite element simulation software for structural mechanics is usually designed following a three-step structure:

1) A graphic interface allows for the interactive design of a geometrical structure and for the definition of finite elements in two or three dimensions. It produces a simple output stream that can be read by a finite element solver.

2) The finite element solver reads information about node location, element assembly, and element types. It further reads initial conditions, external loads, temperature source information, material properties, and other variables. It then solves the system of equations, as a static solution for stationary problems, or as a result of solving of 2nd order dynamics for time dynamic problems.

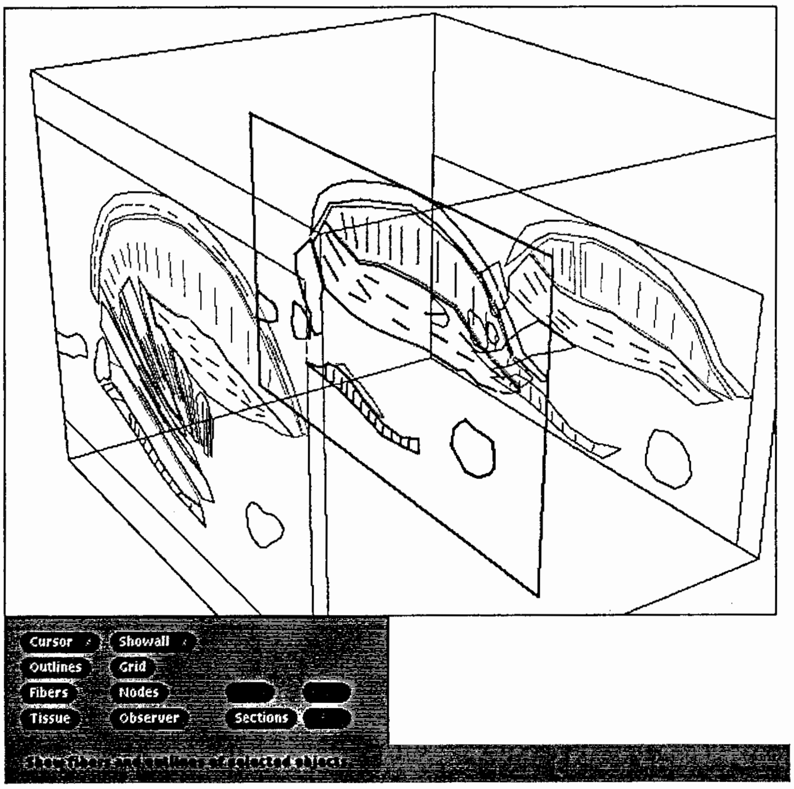
3) The output of the solver, which can be a series of states of the finite element system (in dynamic problems) or a static equilibrium in stationary problems, is read by a third part which brings it to display. This part in many cases includes high performance display of the results, applying a battery of modern visualization techniques.

The above three-steps structure has evolved from the early days of finite element application codes, in which the solution of static problems was predominant. Most dynamic problems in structural mechanics are also successfully solved with this approach because in many cases dynamic problems can be reduced to periodic dynamic problems. In particular, if we are dealing with a linear system (or approximated linear system), any non-stationary input to the dynamic system can be decomposed in linear combinations of a basis of some function space. This allows to solve the general case by superpositioning.

Nonlinear quasi-static (not containing velocities) problems in finite elements, applied to structural mechanics, are often dealing with creep, forming processes, crack formation, heat flow, and others. Even most of these cases are managable using the above three step method, however clumpsy the implementation may be.

Applied to non-linear dynamic problems, the above structure turns out to be a hindrance. Mostly, rather special dynamic problems are being solved, the problems often originating in impact situations where mechanical objects combust on (else where determined) trajectories.

(This study was condected by the author at ATR during the period from April to September 1992.)



Show fibers and outlines of selected objects

Figure 1: Tongue sections displayed with **observer**, an application program built based on the library that is described in this manual.

1.a Requirements in biological system modeling

The deformation of biological tissues during the speech production process is a non-linear and non-stationary problem. The central reason, that commercial FEM packages can not easily be applied to **active** biological tissue deformation, can be found in the particularities of the *stress generating mechanism.* In structural mechanics, active stress generation as by human musculature is a very rare case. Therefore it is not surprising that special software for this purpose is not available to my knowledge.

Another limitation can be found in computer-aided design programs for solid design. If we want to model tissues with finite element simulation methods we need to fully describe element by element, a complex material structure which does not lend itself for solid formation from surface definition. The FEM modeling of biological tissue can be seen as definition of scalar– and vector-valued data on a topological cell structure, where each cell usually corresponds to a finite element. In general, the cell structure itself can not be generated by automatic means, such as automatic grid generation techniques. For these reasons special graphic design tools are required.

The availability of new programming methods lends additional credence to the writing of a new library. The use of the programming language C++, in combination with tools for symbolic mathematics (in the present case Mathematica), allows and demands clearer structuring, true modularity and real data encapsulation. Mathematica is used in the library under development for automatic program generation.

1.b Why C++?

Using the programming language C++, we have all well-known and worthwhile features of C, and at the same time, where algorithms are concerned, a way of writing code that resembles mathematics more then programming code. Several other reasons add to this:

User Interface Programming

The development of this library was started as an extension to a free software library for graphic programming. This software, *The User's Interface Toolkit (UIT)* is written in C++ and contains rather appealing and easily applied class wrappers of XView functionality. In comparison to the confusing, sometimes only partially understood way of programming for the X11 window system under either `Intrinsics` or `XView`, the UIT class library shows a striking simplicity in its class definitions. To illustrate this point, the following example compares a programming example for opening a window and including a canvas inside, first with straight XView programming, and then using the C++ class wrappers in the UIT class library.

Both of the following C code excerpts were originally generated with an interactive user-interface design program called **devguide** (Solaris, SunSoft software). The first code-excerpt is produced with a translator called **gnt** (Solaris, SunSoft software), and the second using a tranlator named **guic** provided in the free software package UIT by Sun Microsystems.

```
/* Create object 'window1' in the specified instance. */
Xv_opaque
Twobuttons_window1_window1_create(ip, owner)
Twobuttons_window1_objects *ip;
Xv_opaque owner;
{
Xv_opaque obj;
obj = xv_create(owner, FRAME,
```

```
XV_KEY_DATA, INSTANCE, ip,
XV_WIDTH, 450,
XV_HEIGHT, 300,
XV_LABEL, "Base Window",
FRAME_SHOW_FOOTER, TRUE,
FRAME_SHOW_RESIZE_CORNER, TRUE,
NULL);
return obj;
}
/* Create object ‘controls1’ in the specified instance. */
Xv_opaque
Twobuttons_window1_controls1_create(ip, owner)
Twobuttons_window1_objects *ip;
Xv_opaque owner;
{
Xv_opaque obj;
obj = xv_create(owner, PANEL,
XV_KEY_DATA, INSTANCE, ip,
XV_X, 240,
XV_Y, 60,
XV_WIDTH, 160,
XV_HEIGHT, 180,
WIN_BORDER, FALSE,
NULL);
gcm_initialize_colors(obj, NULL, NULL);
return obj;
}
main(argc, argv)
int argc;
char **argv;
{
/* Initialize XView. */
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
INSTANCE = xv_unique_key();
Twobuttons_window1 = Twobuttons_window1_objects_initialize(NULL, NULL);
xv_main_loop(Twobuttons_window1->window1);
exit(0);
}
```

The output of UIT's guic looks comparably simpler and much less intimidating:

```
void main (int argc, char **argv)
  BaseWindow window1;
  window1.initUI (argc, argv);
  window1.setWidth (450);
  window1.setHeight (300);
  window1.setLabel ("Base Window");
  window1.show (TRUE);
  window1.setDisplayFooter (TRUE);
  window1.setResizable (TRUE);
```

```
  ComponentDisplay controls1 (TRUE);
  controls1.setX (240);
  controls1.setY (60);
  controls1.setWidth (160);
  controls1.setHeight (180);
  controls1.setDisplayBorders (FALSE);
  window1.addDisplay (controls1);
  Notifier notifier;
  notifier.start ();
  exit (0);
}
```

It is the simplicity of the second programing style which gives the application programmer and the software developer more confidence for developing complex programs. The design of interfaces is already facilitated by the availability of interactive design tools such as the tool **devguide** which was applied. An interactive interface design becomes powerful if it is combined with a code generator, that produces code which can easily be understood and manipulated.

The UIT class library was used as a basis for extension because it is represents a rather insightful hierarchy of graphic objects. In its design, the authors made an effort to present a intuitive hierarchy (see The UIT Technical Overview, p. 9). Extensions can be made in a rather obvious way. Using the C++ class inheritance method, features of class objects of the UIT library were used to build convenient classes which represent for example, a 3D-terminal for polynomial data, and an "observer's eye" (a panel that allows the selection of view-point, observer distance, choice of parallel or central projection etc.). Some of these objects are themselves user interfaces, and `devguide` was used to design their user surface.

Mathematical Programming

In the library some general classes, representing vectors, 2nd order tensors, points, lines, and planes in three dimensions were written. A feature of C++ called *operator overloading* was employed to realize various algebraic operations on the above class objects. Programming in C++ allows the implementation of mathematical operations upon elements other than real numbers as in the following example:

The mathematical expression (v is a 3 dimensional vector, representing a rotation axis direction, and α a rotation angle)

$$\begin{array}{lcll}
w & := & \hat{v} & \text{(normalized)} \\
Id & := & diag\{1,1,1\} & \\
V & := & skew(w) & \text{(Skew symmetric tensor)} \\
W & := & ww^T & \text{(outer product)} \\
T & := & V + (Id - V)\cos\alpha + W\sin\alpha &
\end{array} \tag{1}$$

can be represented by the following C++ code (see class definition of rotator):

```
vector3 w=v.Norm();
tensor3 V(w,w);                                    // generator: ww^T
tensor3 W(w);                                      // generator: skew symmetric
tensor3 Id(1.0);                                   // generator: unit tensor
tensor3 T = V + (Id-V)*cos(alpha) + W*sin(alpha);
```

Fast and still elegant..

The development of class libraries in C++ is still in its infancy. Several attempts to create useful objects for mathematical programing are known. The NIH class library (National Institute of Health Class Library, (USA)) a public domain software [1] has become the best known project of this kind, and it is under consideration to use some of the class definitons. A not so well known second class library called LEDA, written in Germany by Stefan Näher at the Max-Planck-Institut für Informatik, Saarbrücken, is used in the current library. [2] It contains, in particular, class definitions for discrete mathematics, enabling graphs and networks to be easily implemented. Currently the class `matrix` and `vector` for general matrices and vectors of any finite dimension is used.

For the purpose of finite element simulations, where the data structure is a discrete graph, we want to be able to do typical graph operations such as *"for all nodes do ... "*, or *"for all not visited edges do ... "*. Further, we require the implementation of automatic node-renumbering algorithms, which optimize the enumeration of the nodes such that the matrices which result from the finite element formulation have small bandwidths. Realizing this in a language like C++ is a considerably easier task than in other not object oriented languages.

2 Description of the library

At the present state, the library is far from complete. The predominant work was done in implementing convenient classes for user interfacing, whereeas the development of help classes for mathematical objects is only in its infancy. Below a short overview is given with an outline of further develoments.

2.a Linear Algebra operations

In the library, the classes **vector2**, **vector3**, **tensor3**, **Point3**, **Line3**, and **Plane3** are implemented. These objects correspond to 3-dimensional vectors, 2nd order tensors, points, lines and planes. The classes vector3 and tensor3 implement the essential features to write algebraic expressions using the corresponding mathematical elements. In other parts of the software, these classes were applied. To illustrate the use of these classes, the following example was borrowed from the C++ code in the library member **mapper3D**:

Given the following points (members of the class structure mapper3D): an eye-point `EP` (observer's eye point in a three dimensional coordinate system), a view reference point `VRP` (the point which will be mapped onto the center of the projection window), and an up-point `UP` (usually a point on the z-axis); the subroutine calculates an orthonormal system `(U,V,N)` which is used for calcualting the view transformation.

```
void mapper3D::make_transform()
{
   dist = abs(EP-VRP);

   N = norm(EP-VRP);
   V = norm(UP - VRP);
   V = norm(V - ((N*V)*N));

   U = V^N;                  // cross product
   R = vector3(VRP);         // vector version of reference point
}
```

---

[1] NIH is available via anonymous ftp from alw.nih.gov (198.231.128.251) in file pub/nihcl.tar.Z.

[2] At ATR: LEDA is currently installed on hsun23 in the directory /homes/wilhelms/LEDA. A printed manual is available.

Another example using the tensor class is given in the description of the class rotator.

Outline

The definition of the algorithmical components has only been started. For the finite element implemenation, the code for a simple brick element (see shape.h, shape.cc) was written. Further classes that are currently implemented or planned:

> Classes **Element** and **Node** are abstract classes. The classes brick and brick27 are derived from class Element.
>
> An **assembly** is an object which holds the mapping between local node points and global node points. An assembly is a friend class for each finite element. For an element N to obtain it's m-th nodal point information, the assembly is "consulted". The assembly class contains a member function which optimizes the organization of the assembly to provide matrices of minimal bandwidth.
>
> A **Muscle** is a special list that contains references to a collection of nodes in the finite element graph. A muscle can be activated using special member functions which changes the constitutive parameters in a node of the graph.
>
> A **Iterator** is an object that operates on an assembly, solving the system of equations.

2.a.1 Class vector3

The class **vector3** implements the idea of a column 3D vector. Functions that are non-members but friends of the class are marked with the ♡ symbol in the first column.

vector3(double a, double b, double c)		Constructor that defines a vector.
vector3()		Creates a Null vector
vector3(Point3 a)		Makes a type change from Point to vector.
double	Abs()	returns the magnitude of the vector.
vector3	Norm()	returns the normalized vector
void	operator $+=$ (vector3 pl)	Add a vector pl to *this* vector
void	operator $-=$ (vector3 pl)	Subtract a vector pl from *this* vector
double&	operator[ ](int n)	Right or left side, gives reference to $v_n, n = 0, 1, 2$
♡ vector3	operator $-$ (vector3, vector3)	subtract two vectors.
♡ vector3	operator $+$ (vector3, vector3)	add two vectors.
♡ vector3	operator $-$ (vector3)	negate a vector (left operator).
♡ ostream&	operator $<<$ (ostream&,vector3)	print a vector
♡ vector3	operator $\wedge$ (vector3, vector3)	outer product (cross product)
♡ double	operator $*$ (vector3,vector3)	inner product (dot product)

♡ vector3	operator $*$ (vector3,double)	product by scalar from right
♡ vector3	operator $*$ (double,vector3)	product by scalar from left
♡ vector3	operator / (vector3,double)	divide by scalar
♡ tensor3	operator % (vector3 a,vector3 b)	defines a tensor by taking the cartesian product ($a \otimes b = (a_i b_j)$)
♡ double	abs(vector3)	returns the magnitude of the vector
♡ vector3	norm(vector3)	returns the normalized vector

2.a.2 Class tensor3

The class **tensor3** implements the essential algebra for a 2nd order 3×3 tensor. Functions that are non-members but friends of the class are marked with the ♡ symbol in the first column.

tensor3(double a, double b, double c)		Defines a diagonal 2nd order 3 by 3 tensor
tensor3(double a)		diagonal with same value
tensor3()		null tensor
tensor3(double a, double b, double c, double d, double e, double f)		Creates a symmetric tensor with lower triangle given as the six arguments: $\begin{pmatrix} a & b & d \\ b & c & e \\ d & e & f \end{pmatrix}$
tensor3(vector3 a)		Produces a skew symmetric tensor used for rotations around a: $\begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix}$
tensor3(vector3 a,vector3 b)		tensor ab^T, where a and b are column vectors.
tensor3(vector3 a, vector3 b, vector3 c)		tensor made from the three column vectors a, b, c
tensor3 Sym()		returns the symmetrized form of the tensor $\frac{1}{2}(A + A^T)$.
double&	val(int n,int m)	left side or right side to get or set the value. Indices starting at (0,0).
double&	operator () (int n, int m)	Access scalar value directly: $A(i,j), i = 0, \ldots, 2, j = 0, \ldots, 2$
double	Trace()	trace (1st invariant)
double	I2()	2nd invariant

double	Det()	determinante (3rd invar.)
tensor3	Transp()	transposed
tensor3	Inverse()	inverse of 2nd order tensor.
tensor3	InverseT()	inverse transposed of 2nd order tensor.
♡ void	operator $*$ (double f)	Multiplication with double (T*f)
♡ void	operator $*$ (double f, Tensor T)	Multiplication with double (f*T).
♡ double	trace(tensor3)	trace (1st invariant)
♡ double	i2(tensor3)	2nd invariant
♡ double	det(tensor3)	determinante (3rd invar.)
♡ ostream&	operator << (ostream&,tensor3)	print matrix
♡ vector3	operator $*$ (tensor3 &, vector3 &)	tensor times vector.
♡ vector3	operator $*$ (vector3 &, tensor3 &)	vector transp. times tensor.
♡ tensor3	operator $*$ (tensor3 &, tensor3 &)	tensor times tensor.
♡ tensor3	operator + (tensor3 &,tensor3 &)	operator plus
♡ tensor3	operator − (tensor3 &,tensor3 &)	operator minus
♡ double	operator | (tensor3 &,tensor3 &)	inner product
♡ tensor3	transp(tensor3 &)	transposed as nonmember *friend* function
♡ tensor3	inv(tensor3 &)	inverse of 2nd order tensor as nonmember *Friend* function
♡ tensor3	invT(tensor3 &)	inverse transposed of 2nd order tensor as nonmember *Friend* function

The two classes **vector3** and **tensor3** allow to implement simple mathematical expressions in a straight forward way:

For example with A a second order tensor, and I a unit tensor (2nd order), we can have the expression $V = (I - A^T A)^{-1}$:

```
tensor3 I(1.0);                          // diagonal unit matrix
vector3 a(0.1,0.2,0.3),b(1.0,2.0,-1.0);
tensor3 A(a,b);                          // cartesian product a,b
tensor3 V = (I - A.Transp() * A).Inverse();
```

2.a.3 Classes Point3, Line3, Plane3

The class **Point3** is not close to the mathematical definition, it rather follows the practical needs. We just want to have an additional class which has the same data structure as a vector3 object. (Friend functions are marked by a ♡.)

Point3(double x,double y,double z)		Generates a point specifying the cartesian coordinates
Point3()		Generates a point at the origin
Point3(vector3)		makes a point given a vector
♡ vector3	operator − (Point3 a, Point3 b)	Create a vector corresponding to the arrow from point a to point b.
♡ Point3	operator + (Point3,vector3)	translate a point by a vector
void	operator += (Point3,vector3)	translate a point by a vector
void	operator −= (Point3,vector3)	translate a point by a vector
double&	operator [ ] (int n)	reference to nth coordinate, starting at 0.
♡ ostream&	operator << (ostream&, Point3)	print the point to a stream. (E.g. cout << P;)

A simple class **Line3** represents Lines as by a point together with a direction.

Line3()		Create an empty Line
Line3(Point3 o, vector3 t)		Create a line from a point and a direction pointer
Line3(Point3 a, Point3 b)		Create a line from two points, direction is from first to 2nd
vector3	Direction()	returns the direction vector
Point3	Reference()	returns the reference point

A simple class **Plane3** defines a plane in 3 dimensions:

Plane3(Point3 A, vector3 a,vector3 b)	The point A is a point in the plane, a is the first direction and b the second direction. The two vectors are normalized and stored in the object.
Plane3(Point3 A, Point3 B, Point3 C)	The plane is defined by the point A as reference, and the normalized vectors $B - A$ and $C - A$.
Plane3()	A plane at the origin, with the canonical y and z axis as plane coordinate system
Point3 Reference()	returns the reference point

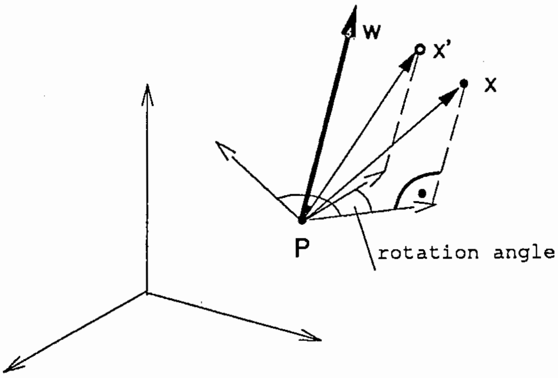


Figure 2: A rotation

vector3	Direction(int *i*)	*i* must be 1 or 2. Returns the first and second vector spanning the plane
vector3	Norm()	returns a vector perpendicular to the plane by taking the cross product of the two vectors that span the plane. The returned vector has a length 1.
Point3	Intersection(Line3 &l)	returns the point of intersection with a line
Boolean	intersects(Line3 &l)	returns a value of True if there is an intersection
Line3	Intersection(Plane3 &p)	returns the point of intersection with another plane
Boolean	intersects(Plane3 &p)	returns a value of True if there is an intersection
double	Distance(Point3 &P)	returns the distance of the Point from the plane
vector3	Project(Point3 &P)	transformes the point coordinates to the plane's coordinate system which consists of the two directions, and the normal vector.
vector3	Project(vector3& *v*)	transforms (via projection) the vector *v* into the coordinate system of the plane. The base vectors are the direction 1 and 2 vectors and the normal vector)

2.a.4 Rotator class

This class represents the rotation transformation around a specified axes by a specified angle. The axis is defined by a point and a unit direction vector. The angle is in degrees and means a rotation in the mathematical positive direction (counter clockwise). The following describes shortly the mathematics behind it:

Let $\mathbf{w} = (w_1, w_2, w_3)^T$ be a vector which represents the direction of the axis of rotation. P is a reference point in the axis. From $\mathbf{w}$ we can build the following two operators:

$$\mathbf{I}-\mathbf{V} := \mathbf{I}-\mathbf{w}\otimes\mathbf{w} = \begin{pmatrix} 1-w_1w_1 & -w_1w_2 & -w_1w_2 \\ -w_2w_1 & 1-w_2w_2 & -w_2w_3 \\ -w_3w_1 & -w_3w_2 & 1-w_3w_3 \end{pmatrix} \tag{2}$$

and

$$\mathbf{W} := \begin{pmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{pmatrix} \tag{3}$$

The operator $\mathbf{I}-\mathbf{V}$ projects a vector $\mathbf{x}$, obtained as the difference between a point X and a the axis reference point P, $\mathbf{x} = X - P$, on the plain perpendicular to $\mathbf{w}$, and $\mathbf{W}$ projects and then turns counter-clockwise by 90^o.

To realize the rotation around $\mathbf{w}$, we thus combine these operators in the following manner:

$$\mathbf{T} = \mathbf{V} + (\mathbf{I}-\mathbf{V})\cos\alpha + \mathbf{W}\sin\alpha \tag{4}$$

where α is the rotation angle. This operator turns any point X by an angle of α around the axis through P in the direction of $\mathbf{w}$.

To realize the rotation as a simple operation on points and geometric objects described by points, we have to implement the following: From each point we subtract the coordinates of P, multiply the resulting vector from left with the matrix T and then add the coordinates of P to the result, obtaining the coordinates of the point X after rotation.

The class has the following public members:

rotator()		default settings: w is (1,0,0), reference point is origin, and angle is 90°.
rotator(Point3 **P**, vector3 **w**, double *angle*)		Complete generator of a rotator.
void	setAngle(double *angle*)	Sets the angle of rotation (in degrees).
void	setAxis(Point3 p,vector3 dir)	Sets the axis of rotation.
void	setAxis(Line3 L)	Sets the axis of rotation from a directed line object.
vector3	getDirection()	Returns the vector w, the direction of the rotation axis.
Point3	getReference()	Returns the point P, the reference point in the rotation axis.
double	getAngle()	Returns the rotation angle in degrees.
Line3	PgetAxis()	Returns the rotation axis as a Line3 object.
vector3	turn(vector3 &X)	rotates a vector: $\mathbf{T}X$.
Point3	turn(Point3 &X)	rotates a Point3 object: $P + \mathbf{T}(X - P)$.
Line3	turn(Line3 &L)	rotates a line by rotating reference point and direction.

Plane3	turn(Plane3 &P)	rotates a plane by rotation reference point and the vectors which span the plane.

The implementation of the rotation is the C++ class **rotator**:

```
class rotator{
private:
  tensor3 T;
  vector3 w;
  Point3  P;
  double  alpha;
  void set_defaults()    {
  w = vector3(1,0,0);
  alpha = M_PI/2.0;           // 90 degrees
  P = Point3(0.,0.,0.);
 }

  void make_T();          // calculate T.
public:
  rotator(){set_defaults(); make_T();}
  rotator(Point3 p, vector3 dir, double angle){P=p;w=dir.Norm(),
       alpha=angle*M_PI/180.0;
       make_T();}
  void    setAngle(double angle)            {alpha=angle*M_PI/180.0;make_T();}
  void    setAxis(Point3 p,vector3 dir)     {P=p;w=dir.Norm();make_T();}
  void    setAxis(Line3 L)                  {P = L.Reference();
                                     w = L.Direction();}// normalized
  vector3 getDirection()                    {return w;}
  Point3  getReference()                    {return P;}
  double  getAngle()                        {return alpha*180.0/M_PI;}
  Line3   getAxis()                         {return Line3(P,w);}
  vector3 turn(vector3 &X)                  {return vector3(T*X);}
  Point3  turn(Point3 &X)                   {return Point3(P + T*(X-P));}
  Line3   turn(Line3 &L)                    {return Line3(turn(L.Reference()),
                                                  turn(L.Direction()));}
  Plane3  turn(Plane3 &P)                   {return Plane3(turn(P.Reference()),
                                                       turn(P.Direction(1)),
                                                       turn(P.Direction(2)));}

};

// in rotator.cc:

void rotator::make_T()
{
  tensor3 V(w,w);     // ww^T
  tensor3 W(w);       // skew symmetric
  tensor3 Id(1.0);    // identity
  T = V + (Id-V)*cos(alpha) + W*sin(alpha);
}
```

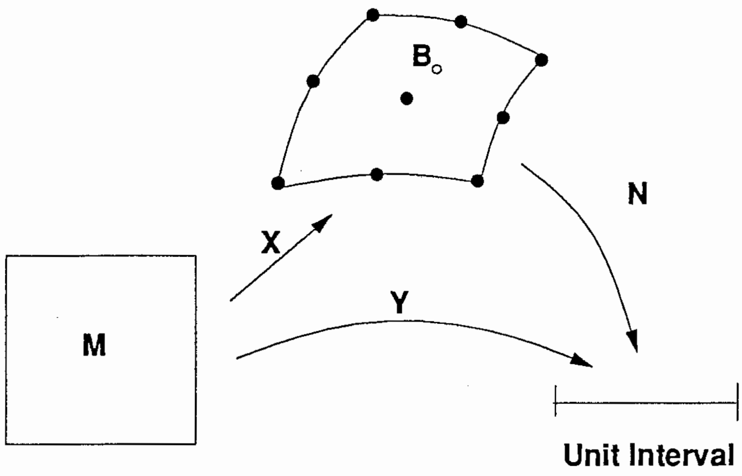


Figure 3: Mappings from the master element to the reference system, and the shape functions defined on the reference system.

2.b Finite element shape functions.

Shape functions are mappings from the domain of a finite element into the real interval [0,1]. They are defined such that they have the value 1 on one of the element‘s node points, and 0 on all others. As an example, the linear brick element has 8 different shapefunctions:

$$\begin{array}{ll} \frac{(1-x)\,(1-y)\,(1-z)}{8}, & \frac{(1+x)\,(1-y)\,(1-z)}{8} \\ \frac{(1+x)\,(1+y)\,(1-z)}{8}, & \frac{(1-x)\,(1+y)\,(1-z)}{8} \\ \frac{(1-x)\,(1-y)\,(1+z)}{8}, & \frac{(1+x)\,(1-y)\,(1+z)}{8} \\ \frac{(1+x)\,(1+y)\,(1+z)}{8}, & \frac{(1-x)\,(1+y)\,(1+z)}{8} \end{array} \tag{5}$$

Here $\{x, y, z\}$ is an element of the cube $[-1,1] \times [-1,1] \times [-1,1]$ For the extended brick element which has 26 nodes (or 27 with a central node) there would be 26 or 27 shapefunctions. The 27 functions for the brick with central node can easily be generated from the outer products of the functions for quadratic interpolation on the interval $[-1,1]$:

$$\frac{(1-x)\,x}{2} \quad , \quad (1-x)\,(1+x) \quad , \quad \frac{(1+x)\,x}{2} \quad , \tag{6}$$

These forms, including their differentials where calculated using the symbolic mathematics program *Mathematica*, and are not printed here because of their length.

2.b.1 Class shape, Brick8shape, Brick27shape

So far there are two different shape function objects designed: **Brick8shape** and **Brick27shape**. Both classes are derived from the base class **shape**, which is described here:

shape(int n)		Generator. Sets the node number
int	numNodes()	returns number of shapefunctions
vector	N(vector3 X)	Returns the list of shapefunction values at the location X in the domain of the element. – *virtual*
double	NK(int K, vector3 X)	Returns $N_K(\mathrm{X})$ – *virtual*
double	DNKi(int K, int i, vector3 X)	Returns $N_{k,i}(\mathrm{X})$ – *virtual*
vector3	DNK(int K, vector3 X)	Returns $GradN_K(\mathrm{X})$ – *virtual*
matrix	DN(vector3 X)	Returns the matrix of all gradients of all shape functions, with the gradients as line vectors – *virtual*

The other two classes **Brick8shape** and **Brick27shape**, have the same members, and are directly derived from the base class **shape**. The class **shape** is an abstract class and can not be realized. This implementation is used to allow for programming were the internal structure of the element is not relevant. Which of the functions is called is decided automatically at runtime.

2.c User Interface classes

In the UIT class library which was used as a starting base for the development of further user interface objects, all objects are derived from a class called **Generic**. Generic is an empty class, it has nothing but a constructor and destructor in it. Directly derived from **Generic** are the classes **GenericList**, **GenericHash**, **InputEvent**, and **UIObject**. These form the basis for all other classes that are used for interface programming. GenericList is a simple general list, that also allows access to its members via the [ ] operator. It can hold any members which are class objects derived from the class **Generic**. The class **UIObject** implements the essentials of XView programming, and encapsulates a lot of functionality without demanding a deeper understanding of XView programming. An UIObject has a representation on the computer screen, and can receive XView events.

For the purpose of this software, the top class of the UIT library, **Generic**, was slightly modified and contains an additional member *virtual void receive_other(Generic * g)*, which is used in derived classes for communication between various objects, see below under communication between objects.

The existing class hierarchy was used as a starting basis to create new objects by extending the given class definitions. The resulting hierarchy is depicted in figure 4.

2.c.1 DisplayG

Class **DisplayG**. Inherits directly from **ComponentDisplay**, and indirectly from UIDisplay and UIObject.

Public Members:

DisplayG()	Constructor: sets internal defaults

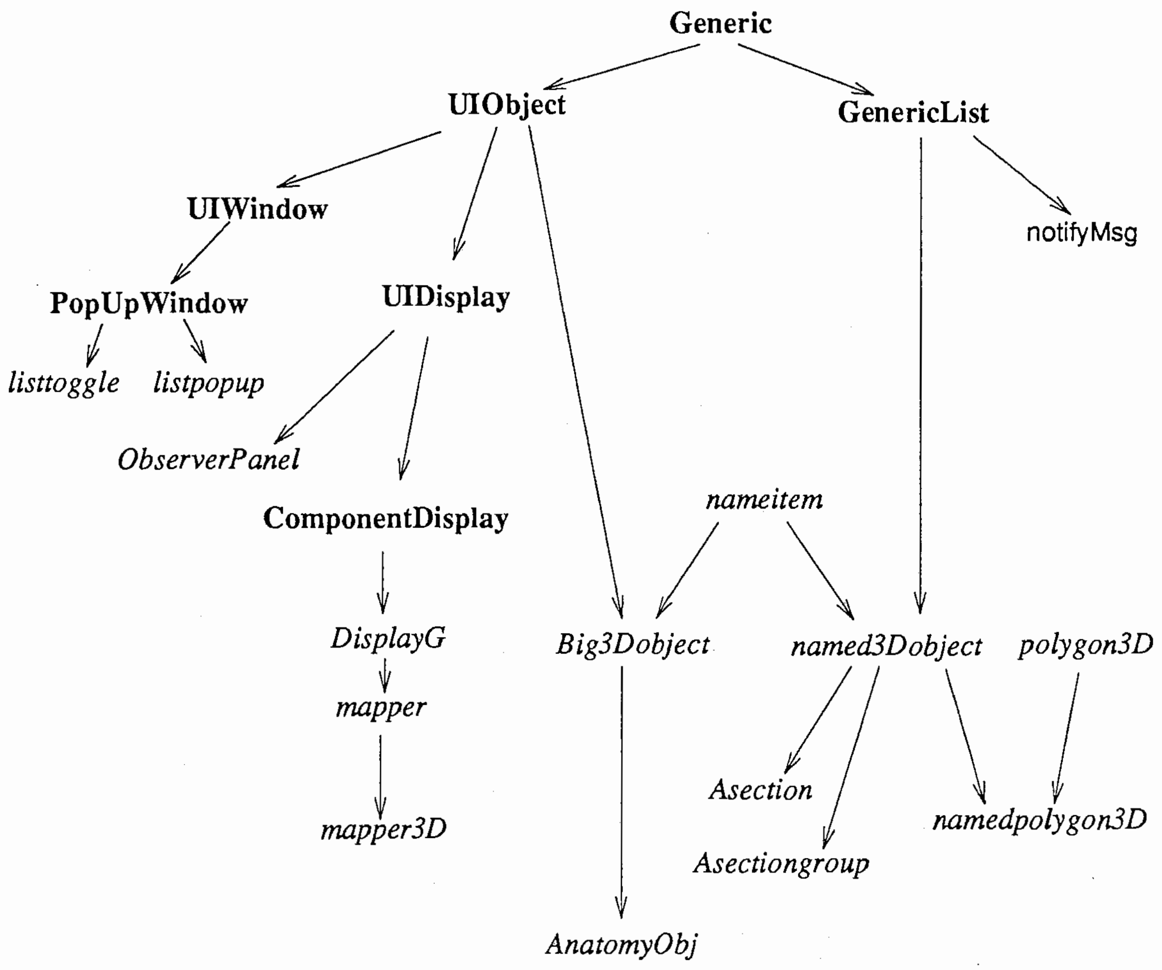


Figure 4: The hierarchy of objects which are extensions of the UIT class library. Classes taken from the UIT library are written in bold, and new classes, derived from the UIT classes, are in italic.

void	receive_other(Generic *)	Communicates with other objects, using notifyMsg class objects. Using the repaintMSG message causes it to repaint by calling its RedrawHandler
void	ClearCanvas()	use X11 call to clear the drawing surface
GenericList*	getPaintList()	returns the list containing objects to paint
void	addgraphicObject (Generic *)	Any graphic object which is derived from a Generic class can be inserted in the list
DisplayG&	operator << (char *colorname)	Used to set a color (like writing the string to standard output with the << operator)
DisplayG&	operator << (long color)	Set a color by pixel value
DisplayG&	operator << (linestyle l)	set a linestyle: Objects of the following classes can be used: dotted_line, dotdash_line, dashed_line, dotdotdash_line, and dash3dot_line. See attributes.h for definition of linestyle classes.
DisplayG&	operator << (graphicoperation g)	A graphicoperation is a class that is for instance created as follows: graphicoperation gob(Xxor). G << gob would then cause the graphics mode to be xor.
DisplayG&	operator << (x11polygon p)	plot a 2D polygon of XPoints, see x11polygon.h
DisplayG&	operator << (XPoint X)	Plot a XPoint
void	setHighlighted (Boolean t=TRUE)	Causes the line thickness to be set to 2, if there is no argument or if it is TRUE, otherwise to 1, which is default.
long	getPixelbyStructureName (char *name)	The program first tries to communicate with all other programs to translate the character string name into a color name. If a listpopup class object is associated with this object the name can be translated, if it is in the listpopup's table. If the name could be translated into a colorname, it is translated into a pixel value using the default colormap. If the color can not be found, a pixel value of 0L is returned.
long	getPixel(char *colorname)	The default color map is used to find the pixel value
long	getBlackPixel()	returns the pixel for black from default colormap.
long	getWhitePixel()	returns the pixel for white from default colormap.

2.c.2 mapper and mapper3D

Class **mapper**. Inherits directly from **DisplayG**, and thus indirectly from ComponentDisplay, UIDisplay, UIObject.

Members:

double xscale, yscale,virt_height, virt_width,virt_origx,virt_origy		*protected.* These data items hold the description of the virtual screen which is mapped onto the drawing surface. There are more protected members, see the include files.
mapper()		Constructor. Sets defaults.
void void double double	setWorldHeight(double); setWorldWidth(double) getWorldHeight() getWorldWidth()	These public functions are right now implemented to access protected data. The class mapper was primarily written as a baseclass for mapper3D, which can directly use the protected class members. So these functions may be obsolete for the mapper class.
void	moveto(vector2 &v)	Moves to the point without drawing.
void	drawto(vector2 &v)	Moves to the point drawing.
void	drawpolygon(vector2 *v, int n)	Draws a polygon of n points, moving to the first without drawing.
void	drawme(mapper &m)	Empty function.

The class **mapper3D** inherits directly from **mapper**, and thus indirectly from DisplayG, ComponentDisplay, UIDisplay, UIObject.

Public Members:

mapper3D()		Constructor
void	renewParameters()	This member communicates with other objects that are connected to this object in order to obtain a viewpoint information. The notifyMsg of type viewpointREQMSG is sent to all associated objects. (see ObserverPanel, FlowNode.h)
void	set_vrp(Point3 vrp)	Sets the view reference point explicitely
void	set_vrp(double x, double y, double z)	Sets the view reference point
void	set_ep(Point3 ep)	Sets the position of the observer's eye-point.
void	set_ep(double x, double y, double z)	Sets the position of the observer's eye-point.

void	set_up(Point3 up)	Sets the up-direction. If not set the z axis is used as default.
void	set_ep_polar(double height, double length, double dist)	Sets the observer's eye point in spherical coordinates. height is a value between -90.0 and +90.0 describing the altitude of the observer's eye. 0.0 corresponds to the horizon which is the x-y plane. length is a value between 0.0 and 360.0 describes the longitude, whereby 0.0 corresponds to the direction of the x axis. dist is the distance of the eye-point from the view reference point.
void	set_observer_height(double h)	Set altitude of observer
void	set_observer_distance(double d)	Set observer distance
void	set_observer_length_angle(double l)	Set observer longitude
void	set_observer_viewangle(double angle)	This angle in degrees (common values 2-45) determines the viewing width of the mapping. A small value results in a mapping that is close to parallel projection.
void	setparallelProjection (Boolean t=TRUE)	Logical switch between parallel and perspective projection
Boolean	Projection_is_parallel()	Check the value of the switch which determines if the projection is parallel or perspective
vector2	project(vector3)	The function that allows to calculate a mapping without drawing anything. Returns the 2D vector that could be plotted by an object of class mapper to produce an image on the screen.
XPoint	screen_map(vector3 x)	returns the XPoint after projection, considering the current window size.
void void	moveto(vector3 &v); moveto(Point3 P)	Calculate the projections on the screen and move to the corresponding point or vector without drawing a line.
void void	drawto(vector3 &p); drawto(Point3 P)	Move to point or vector with drawing a line from the previous position. Last point moved to is always memorized in the mapper.
void	drawpolygon(vector3 *v, int n)	Draws a 3D polygon as projection on the screen. Moves to the first point without drawing.
void	drawme(mapper &m)	Generates a simple colored coordinate cross. *This function is not implemented – see ObserverPanel class, which contains the drawing of a little coordinate axis system.*

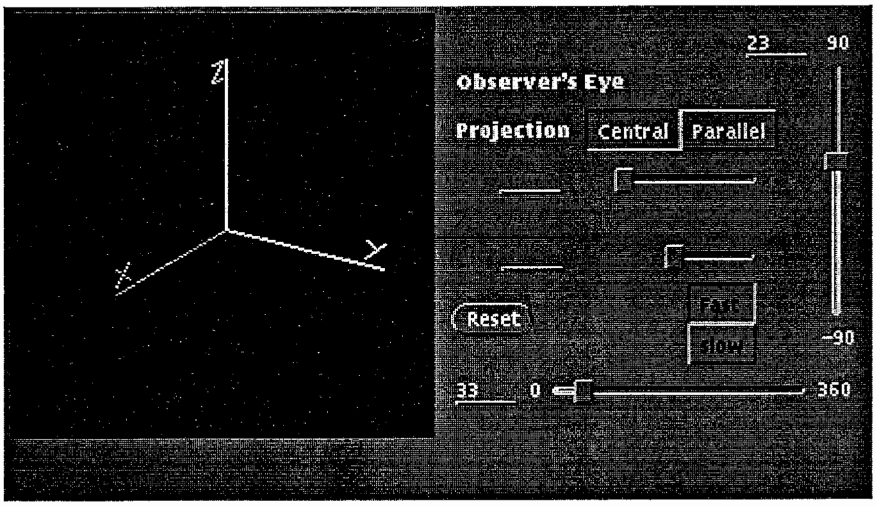


Figure 5: Appearance of a class object ObserverPanel

2.c.3 Linestyle classes in attributes.h

In the file attributes.h some classes are defined which implement such concepts as a dashed line or a color. These class definitions are mostly obsolete. Only the definitions for linestyles should be used.

linethickness and dotted_line are classes that creates an object as follows:

```
linethickness lt(2);
dotted_line dl;
```

This creates an object representing a linethickness of 2 and one representing a dotted_line. They can be seen as symbols to be written to a device to put it in a particular mode so that it uses thick dotted lines: The objects can be be written to the mapper3D object or a DislayG object in the following way:

```
*M3 << lt << dl << ''Green'';
```

where M3 is a pointer to a mapper3D object. After that drawing will be done in thicker green dotted lines.

Other classes of this kind are the following: **dotdash_line,dashed_line**, **dotdotdash_line**, and **dash3dot_line**

Colors as long integers (representing a pixel) can be treated the same way as explained for colorstrings.

2.c.4 ObserverPanel

The object **ObserverPanel** is directly derived from **UIDisplay**. It contains a drawing surface which is an object of type mapper3D, and a control panel (object of class ComponentDisplay) which contains four sliders and some buttons. The sliders control the altitude, longitude, distance and viewing width of an observer's eye-point. One button switches the projection type between parallel and perspective projection. A switch allows the selection of slow or fast mode. In fast mode a redraw signal is sent to all associated objects after any change of one of the parameters, and in slow mode a pushing the set button is required (only visible in slow mode). The reset button will set the system to predefined (see ObserverPanel.h) values. See figure 5 for a the outlook of the object when implemented within a popup window.

This class contains a set of internal (private) handler functions which allow it to react to events occuring on the sliders and buttons. **NOTE:** An object of class ObserverPanel has to be created in a BaseWindow or in a PopUpWindow. Both its mapper3D panel (with the rotatable coordinate system) and the control panel have to be made child objects of the parent BaseWindow or PopUpWindow. To accomplish that, a call of ObserverPanel::connect2base() is required.

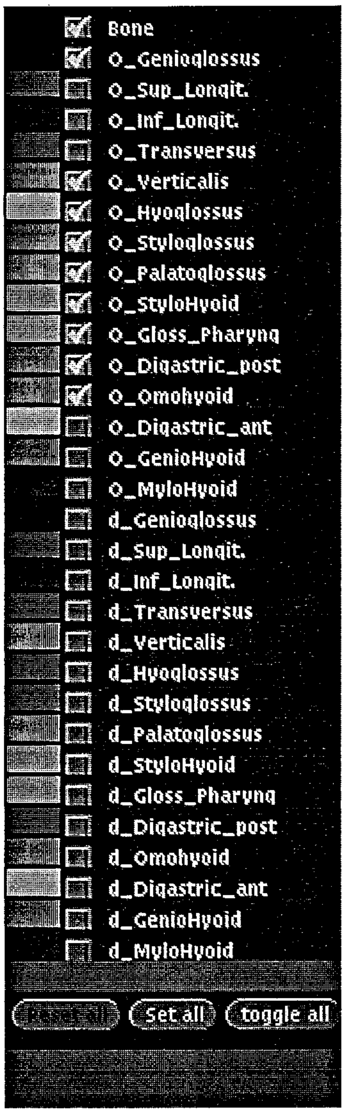
OberserverPanel()		The generator. It installs all the sliders, buttons and toggles within the object and initialized them.
int	get_observer_height()	returns observer altitude (between -90 and +90)
int	get_observer_width()	returns viewing angle in degrees.
int	get_observer_horiz()	returns observer longitude (0 to 360 degrees)
int	get_observer_dist()	returns observer distance.
Boolean	is_Projection_parallel	returns TRUE for parallel projection or FALSE for central projection.
void	set_obs_height(int h)	sets observer's altitude (-90 to 90 degrees)
void	set_obs_width(int w)	sets observer's viewing angle (1-45 degrees)
void	set_obs_hor(int h)	sets observer's longitude (0-360 degrees)
void	set_obs_dist(int d)	sets observer's distance (1-999 units)
void	setX(int x)	sets left upper corner X coordinated in the application window
void	setY(int x)	sets left upper corner Y coordinate in the application window
void	connect2base(BaseWindow *BWind)	Use this function after creating an ObserverPanel to make its canvas and its control field children of the BaseWindow
void	connect2base(PopUpWindow *PWind)	Use this function after creating an ObserverPanel to make its canvas and its control field children of the PopUpWindow

2.c.5 listpopup

The class **listpopup** is derived from **PopUpWindow**, and thus indirectly from UIWindow and UIObject.

The purpose of the class is to show a list of names that are associated with colors. For each name the color is shown in a little rectangle. The user can select and deselect a check marker for each name. A listpopup object communicates notifyMsg's of the type selectMSG and unselectMSG and redrawMSG to all connected objects. There are three buttons which allow general settings in the list.

Public Member functions:



☑ Bone
☑ O_Genioglossus
☐ O_Sup_Longit.
☐ O_Inf_Longit.
☐ O_Transversus
☑ O_Verticalis
☑ O_Hyoglossus
☑ O_Styloglossus
☑ O_Palatoglossus
☑ O_StyloHyoid
☑ O_Gloss_Pharyng
☑ O_Digastric_post
☑ O_Omohyoid
☐ O_Digastric_ant
☐ O_GenioHyoid
☐ O_MyloHyoid
☐ d_Genioglossus
☐ d_Sup_Longit.
☐ d_Inf_Longit.
☐ d_Transversus
☐ d_Verticalis
☐ d_Hyoglossus
☐ d_Styloglossus
☐ d_Palatoglossus
☐ d_StyloHyoid
☐ d_Gloss_Pharyng
☐ d_Digastric_post
☐ d_Omohyoid
☐ d_Digastric_ant
☐ d_GenioHyoid
☐ d_MyloHyoid

Figure 6: Appearance of a class object of type listpopup. It reads the list of structures and colors to display from a file. The class allows selection and deselection of named items in the application.

listpopup(char *file, char *title)		Class constructor. The file has to contain items of the kind: `structure-name color` and is has to contain the number of list entries in the first line. See the file anatomytable.map as an example
~listpopup()		Class destructor
Boolean	isSelected(int l)	returns TRUE if line l (starting at 1) is selected, otherwise FALSE
Boolean	isSelected(char *name)	returns TRUE label name exists and is selected, otherwise FALSE
void	setSelected(int l)	set the label in line l selected
void	setUnselected(int l)	set the label in line l unselected
char*	fieldlabel(int n)	returns the character string of the label of line n (starting at 1)
int	NumberofLabels()	returns the number of list items.
char*	color_of(char *name)	translates a structure into a colorname according to the that was used to create this object.
void	communicate(UIObject *o)	function used by this object to send information to other objects (should be private).
void	receive_other(Generic *g)	function to receive information from other objects. This function is virtual in the base class **Generic**

2.c.6 listtoggle

Similar to **listpopup** the class **listtoggle** is derived from **PopUpWindow** and thus inherits indirectly from UIWindow and UIObject.

This object is like a two-dimensional list. For a number of names several states can be selected in an exclusive choice. For example, a list of named graphical objects can have an exclusive state variable which can have the state invisible, visible, highlighted. The construction of the class is the only complicated part of it: One specifies a list of names, and another list of type MsgType (as specified in **FlowNode.h**)). Selection of one of the choices in a list element causes that a listtoggle class object sends a message to all associated objects and specifies the as MsgType the type that is associated with the pressed button's column.

Public Member functions:

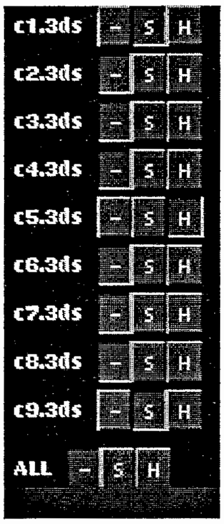


Figure 7: This is an object of the class listtoggle. It allows selection from a list of mutually exclusive states of objects.

listtoggle(char *title, int nlines, char **labels, int mcolumns, char **entries, MsgType *messages, Boolean master=FALSE)		Class constructor: The title appears in the title field of the underlying PopUpWindow. nlines gives the number of lines (labels). labels is an array of character strings which become the labels for each line. mcolumns is the number of columns associated with each line. entries is an array of (length mcolumns) of labels to be placed on the bottons in each line. messages is an array (length mcolumns) of variables of type MsgType which specify what message type has to be sent if one of the columns in a line is selected. master is a variable which determines if there should be a general line which allows everything to reset.
~listtoggle()		Class destructor
int	Selection(int n)	returns the column number of the selection in a line n
int	Selection(char *name)	returns the column number of the selection in the line with the label name
void	setSelection(int line,int column)	sets one entry selected and the other in the same line unselected
char*	fieldlabel(int n)	returns the field label of line n
int	NumberofLabels()	returns the number of lines
void	setMasterSettings()	This is usually only used internally; it should be a private member function.

2.d Graphic objects

Class **named3Dobject** is a derivative of the class **GenericList**. From GenericList it inherits all the features of a fast operating list of Generic objects. As long as all graphic objects are derived from the base class Generic, they can be assembled to any complexion of graphic objects. Unfortunately, the class UIObject would even be better as a base class for graphic objects, since it then could also receive Xview events, could have it's own event handler, and other useful features of UIObject. The following considerations lead to the decision of having two types of objects for 3D graphics, the named3Dobject and the Big3Dobject:

named3Dobject: There is a lot of special simple objects like for instance polygon, or rectangle. However, a simple polygon should have some more features, as there are: it can be hightlighted, it can be translated, it can be rotated. It can have a name which decides over its being selected for display or not, etc. Since we want to have thousands of polyugons in an application, one of them shouldn't take too much space away. If each graphic object of this kind would be based on the class UIObject, the storage overhead would be considerably high.

Big3Dobject This class should have mostly the same features as the **named3Dobject** but should further be able to communicate as a normal graphic object in the UIT class library. For example, HotRegions can be associated with it enabling direct interaction with the mouse and keyboard, and it can be a target of Xview events (and thus X11 events). An important reason for installing this class was that the UIObject class contains the member functions to associate different UIObjects with each other: *setObjectData(), traverseObjectData()*. Further, a Big3Dobject is designed to be loadable from a file (or multiple files), and can be saved to a file (not yet implemented). These functions have to be implemented in the subclasses derived form this class, since they depend completely on the data structure. Thus *Big3Dobject* is an *abstract class.* A Big3Dobject (a class derived from it) will usually contain multiple parts which are objects derived from the class named3Dobject. Usually only few (1 or 2 or more) Big3Dobject are operated within an application program.

In the current project, the program *observer* loads sketches of slices of tongue specimen from multiple files and displays them in 3D. The individual polygons within each sketch of a tongue section are *namedpolygon3D* objects which is a special class derived from named3Dobject. Many such namedpolygon3D objects are in one section, which is represented by the special class *Asection*. The whole set of sections is represented by a class *Asectiongroup*. All three special classes, namedpolygon3D, Asection, and Asectiongroup, are derived from named3Dobject. However, the object *AnatomyObj* is derived from Big3Dobject and contains objects of types namedpolygon3D, Asection, and Asectiongroup. The AnatomyObj contains the whole set of namedpolygon3Ds, Asections, and Asectiongroups.

2.d.1 nameitem

The data of this class contains four items: a char-string as a name, and three Boolean variables, *visible, selected, highlighted.*

Public functions are:

nameitem()	Empty class constructor, generates a nameitem with no name, sets visible to TRUE, the other Boolean variables FALSE
nameitem(char *name,Boolean selected=TRUE)	class constructor: Sets selected to the value sel (default TRUE) and generates a copy of the name character string.

void	setname(char *x)	replaces old name by new name
char	*Name() const	returns a char * reference to the name.
Boolean	hastheName(const char *x) const	compares this name with own name and returns TRUE if they are the same, otherwise FALSE.
Boolean	isVisible()	returns visible
void	setVisible(Boolean v = TRUE)	Sets visible (default: to TRUE)
Boolean	isHighlighted()	returns highlighted
void	setHighlighted(Boolean v=TRUE)	Sets highlighted (default: to TRUE)
Boolean	isSelected()	returns selected.
void	setSelected(Boolean v=TRUE)	Sets selected (default: to TRUE)
Boolean	isNamed()	returns TRUE if the name is set (depends on how the object was created).
void	select_by_name(char *name, Boolean v=TRUE)	Sets selected if the name fits.
void	unselect_by_name(char *name)	Sets unselected if the name fits.
void	highlight_by_name (char *name, Boolean v=TRUE)	Sets hightlighted if the name fits.
void	setvisible_by_name (char *name, Boolean v=TRUE)	Sets the object's variable visible if its name is the same as the name in the argument.

2.d.2 Abstract Class named3Dobject

named3Dobject has multiple inheritance from **nameitem** and **GenericList**. However, the original members *addItem* and *traverse*, which are already defined in the **GenericList** class, are modified such that only other **named3Dobjects** and thereof derived classes can be hung in the list of a **named3Dobject**. Thus, a **named3Dobject** can contain several recursive member functions which do the same operation on all other **named3Dobjects** which are contained in one **named3Dobject**. This class is abstract, that is it contains some members which are not defined and have to be defined in derived classes: make_hull(), translate(), rotate(), paint3DThis().

named3Dobject(char *name)	Class constructor, initializes also nameitem part. Sets the color to non defined value.
named3Dobject()	Class constructor for a named3Dobject which has no name.

virtual void	make_hull() = 0	*Protected* This member has to be defined in classes that are derived from this class. The function returns a Hull object, which is a quarder that contains the graphic object of the derived class. The hull quarder is stored in the named3Dobject class, but the function to create it can not be written as a universal routine, because it depends on the specific properties of the derived special named3Dobject.
void	addItem(named3Dobject *n3o)	Inserts an object of class named3Dobject or a class object derived from named3Dobject. It uses the GenericList part but only allows named3Dobjects to be inserted in the list. This is done by casting the pointer n3o to a named3Dobject pointer, which is a valid operation only if n3o points to an object derived from the named3Dobject class.
named3Dobject	*traverse(Boolean flag)	Traverses the list of named3Dobjects that have been added using addItem() (above), to start the list traversing, the flag has to be set TRUE. By multiply calling this member function while setting flag to FALSE after the first call, one can get a pointer to all named3Dobjects in the list. When the list is exhausted, a NULL pointer is returned.
vector3	center()	Calculates the hull of the object and returns a vector3 containing the center of the hull. The hull is a quarder which contains the whole object.
virtual void	translate(vector3 &v)	Abstract virtual - not defined. Must be defined in derived classes
virtual void	rotate(rotator &r)	Abstract virtual - not defined. Must be defined in derived classes
virtual void	paint3DThis(mapper3D *g)	Abstract virtual: not defined
void	drawHull(mapper3D *m)	Draws a quarder in the 3D mapper which represents the box that contains the object.
void	setcolor(long cc)	cc is the Pixel value which can be obtained from a colormap
long	getcolor()	returns the currently set pixel value of this object.
Hull	getHull()	makes a new hull and returns it.
void	select_all_named(char *n)	*recursive*: this and all objects in this class with the name n are set selected

void	unselect_all_named(char *)	*recursive*: this and all objects in this class with the name n are set unselected
void	highlight_all_named(char *name, Boolean flag)	*recursive*: this and all objects in this class with the name n are set highlighted.
void	setvisible_all_named(char *name,Boolean flag)	*recursive*: this and all objects in this class with the name n are set visible

2.d.3 Class Big3Dobject

is an **abstract class**. It inherits from **UIObject** and **nameitem**. Most of its functions have to be implemented by derived classes since they are data dependent. This class was introduced so that a larger and more complex graphic object is based on the class UIObject rather than on GenericList. This allows the treatement as a point of communication withing the application. See the class **AnatomyObj** which is derived from this one, as an example.

	Big3Dobject()	class constructor: name not defined
	Big3Dobject(char *name)	class constructor: name defined and visible set to TRUE.
	~Big3Dobject()	Class destructor: empty function
virtual void	paint3DThis(mapper3D *M) =0	Undefined, Abstract Class
virtual void	load_from_disk(char *filename) =0	Undefined, Abstract Class
virtual void	save_to_disk(char *filename) =0	Undefined, Abstract Class
virtual void	initialize() =0	Undefined, Abstract Class

2.e Special Classes

2.e.1 Class Hull

This class implements the features of quarders in 3D that contain a 3-dimensional object. A hull is defined by the two 3-dimensional vectors which describe 3 intervals, and can be seen as diagonally opposed edges of the quarder. The first vector contains the lower limits of the intervals and the second the higher values. A hull with no defined limits is empty. The class **Hull** implements an addition operator where the sum of two hulls is the hull which contains both.

	Hull()	Class constructor. Creates an empty hull
	Hull(vector3 v)	Class constructor. Creates a nonempty hull (quarder) containing only one point
	Hull(vector3 v1,vector3 v2)	Class constructor. Creates a quarder
	Hull(double x1, double y1, double z1, double x2, double y2, double z2)	Class constructor. Creates a quarder with explicitely stating the two corners.
Boolean	is_empty()	returns TRUE if the quarder is empty.

vector3	center()	returns the middle point of the quarder and a null vector if it is empty
vector3	lower()	returns the vector with the lower interval limits
vector3	higher()	returns the vector with the higer interval limits
void	clear()	Makes the hull an empty hull
void	operator += (Hull h)	add a hull h to this hull. The result is replacing this hull by a hull which contains both this hull and h
Hull	operator + (Hull a, Hull b)	returns a hull which contains both hulls a and b. *Friend function*

2.e.2 Class polygon3D

polygon3D()		Class constructor. empty polygon
polygon3D(int n)		Class constructor. Space for n edge vectors
polygon3D(Point3 a, Point3 b)		Class constructor. Creates a polygon of two vectors – a straight line
~polygon3D()		Class destructor. Clears memory
void	addpoint(Point3 &X)	append a vector to the polygon
Point3&	operator [ ] (int m)	allows access to the points of the polygon like in an array. Can be used as right hand side of an expression or as left hand side of an expression. If m is out of range, access to the first (index 0)
void	translate(vector3 &v)	Add vector v to each point in the polygon.
void	rotate(rotator & r)	Rotates all points around a reference axis with specified angle. See the class **rotator**
Point3	*first()	returns the address of the first point in the array.
int	no_points(),*or* int cardinality()	returns the number of points in the polygon

2.e.3 Class namedpolygon3D

This is a simple extension of polygon3D, inheriting from **polygon3D** and **named3Dobject**

namedpolygon3D()	Class constructor. Creates nameless, empty polygon
namedpolygon3D(int n)	Class constructor. Creates nameless polygon with space for n vectors

namedpolygon3D(char *na,int n=0)		Class constructor. Creates named polygon with space for n vectors, default if n is not specified: 0
namedpolygon3D(char *na,Point3 a, Point3 b)		Class constructor. Creates named polygon consisting of two vectors
void	paint3DThis(mapper3D *M)	The polygon draws itself with this function.
Point3&	operator [] (int m)	allows access to the member vectors in an array style.
void	translate(vector3 &v)	add v to all points of the polygon
void	rotate(rotator & r)	Rotates all points around a reference axis with specified angle. See the class **rotator**
void	make_hull()	Protected. The hull is stored in the named3Dobject-part of this object. It is defined here, abstract in named3Dobject

2.e.4 Class Asection

This class represents a tongue section sketch. It is derived from the class **named3Dobject**.

Asection()		Class constructor. Empty
Asection(char *name)		Class constructor. A named section (The file name can be used as the name.)
~Asection()		Empty class destructor.
void	load_from_file(char *filename,int x_coordinate)	Loads the section from file. The x and y coordinates in the file are used as y and z coordinates. The new x value has to be specified to obtain a 3D representation
void	paint3DThis(mapper3D *g)	Draw this section by mapping it with the associated mapper3D object
void	translate(vector3& v)	add a vector v to all vectors in the section
void	rotate(rotator& v)	rotate the section with an operator defined in the rotator class object r, see class rotator.
void	make_hull()	*Protected.* Calculates a new hull which is stored in the named3Dobject part

2.e.5 Asectiongroup

This class represents a series of tongue section sketchs. It is derived from the class **named3Dobject**. It contains all data of a series of tongue sketches.

void	make_hull()	Protected. calculates a new hull which is stored in the named3Dobject part
Asectiongroup()		Class constructor. Empty
Asectiongroup(char *name)		Class constructor. A named section group.
~Asectiongroup()		Empty class destructor.
void	add_section(Asection *as)	Adds a section of class Asection
void	add_section (char *filename,int x)	Creates a section by reading it from a file, using the x value to make a 3D slice, and adds it to the list of subobjects.
void	paint3DThis(mapper3D *g)	Draws everything
void	translate(vector3 v)	In each slide, in each polygon, the vector v is added to each vector.
void	center_at_origin()	calculates a new hull and its center, and the translates the whole object to the center of the hull and calculates a new (translated) hull.

2.e.6 AnatomyObj

This class is derived from **Big3Dobject**. It is used as a wrapper to have the tongue sections represented as a Big3Dobject. It can thus communicate with other objects derived from UIObject. The class contains a pointer to an object of class **Asectiongroup** to hold the data.

AnatomyObj()		Empty class constructor
AnatomyObj(char *name):Big3Dobject(name)		Class constructor with name
AnatomyObj(char *name,Asectiongroup &A)		Class constructor with name and reference to already existing Asectiongroup object.
AnatomyObj(char *name,Asectiongroup *A)		Class constructor with name and pointer to already existing Asectiongroup object.
~AnatomyObj()		Class destructor, deletes all data
void	receive_other(Generic *g)	Communication with other UIObjects (see FlowNode.h). Used for allowing highlighting, selection and deselection of parts of the tongue section display.
void	paint3DThis(mapper3D *M)	Calles the painting function in the Asectiongroup (pointed at by the pointer data.
void	load_from_disk(char *filename)	The filename contains a list of other filenames together with a x values that give the position of the section.
void	save_to_disk(char *filename)	Empty: Not yet implemented.
void	initialize()	Currently: Centers around the midpoint of the sections.

2.f class notifyMsg for communication between different objects

From the UIT library comes the concept to link different objects at run time using the UIObject's member function `setObjectData`. `setObjectData` is a member of the class **UITObject**, therefore each class derived from it has this member. For example, in the main program `observer_ui.cc`, the following connection is established:

```
Maincanvas.setObjectData("ObserverPanel",Observer_Panel);
```

As a result in the code for the object Maincanvas (which is of class type mapper3D), a pointer to the object Observer_Panel can be obtained by using the inverse function `getObjectData`.

Various objects as part of this software need to interact in some standardized way. However, the mentioned type of linking objects seemed to be too specific. We don't want to rely on special names of other objects when writing the code for a particular class. All that is needed is a connection between objects and methods of communication. For this purpose, a particular kind of message system, which can be seen as a broadcast message system, was desiged.

For example, a certain object of class XY changes program parameters that require redrawing all views of a graphic object. Since there may be all kinds of drawing routines in the program, XY can broadcast simply: "To whom it may concern: you probably need to redraw." This type of broadcasting messages is implemented in the class **notifyMsg**, which is coded in the files FlowNode.h and FlowNode.cc. It is realized as traversing the list of associated objects and sending them a message. *An object B is only notified by an object A if it is connected with A by a command like:*

```
A.setObjectData(''some string'',B);
```

The method of message sending can be understood from the include file FlowNode.h, where the data type `notifyMsg` is defined, and from the example of an ObserverPanel class object communicating the change of the observer position to other objects, in particular, tt mapper3D class objects. Another simple example is the implementation of the class function `DisplayG::getPixelbyStructureName(char *name)`. The function creates a notifyMsg with the general request to translate a string. The message is sent to all objects that are linked with the DisplayG object. If there is an object that understands this message (listpopup, see the implementation of the function listpopup::receive_other) the receiving and answering object will set a flag in the message structure, notifying back that the request was answered.

The efford of builing inter-object communication in this way is justified by the advantage that interdependencies between the different class definitions can be avoided. If the class DisplayG (and derived classes mapper and mapper3D) would directly call public members of listpopup, the class DisplayG would depend on the class listpopup. This strategy also cuts down on the number of included include-files.

Where the re-used software of UIT class library is concerned, the outlined communication method required only a little modification in the highest base class for all objects in the UIT library: The function *virtual void receive_other(Generic *g)* was included as a member in the class definition of **Generic**. In class **Generic** the function *receive_other(Generic *g)* does nothing but return, whereas in some derived classes the function can be implemented to realize the particular communication abilities of that class.

The communication method is illustrated in the below example.

avoid that each object has to contain very much information about other objects. This requires some general purpose interface between the different objects. A principle of broadcasting was used to realize communication between objects. For example, if a user sets in an object of class **listpopup** certain names as selected, other objects, e.g., a graphics object derived from the class Big3DObject, has to be automatically informed about this. While writing the code for the listpopup class, we want to do the design without any knowledge of future objects which are going to interact with a listpopup class object. To realize that, a listpopup object has to send a message to all other objects that are known to it. Here is an example:

In the main program, somewhere a Big3DObject class object is created, and a listpopup class object. In order to allow communication between them the two have to be connected:

```
main( ...
...
Big3DObject xbach;
...
listpopup  lpop;
...
lpop.setObjectData(‘‘some3Dbigthing’’,xbach);
...
```

In the listpopup class object, the code to communicate looks as follows: (`myself` is a pointer to the class object – the shown code is part of a static handler function)

```
  MsgType mst;
  if (choice) mst = selectMSG;
  else        mst = unselectMSG;

  notifyMsg msg(myself,mst);
  notifyMsg repaintreq(myself,repaintMSG);

  msg.send_to_all(myself,myself->fieldlabel(index),mst);
  repaintreq.send_to_all(myself);
```

Big3DObject can accept the two types of messages selectMSG or unselectMSG:

```
switch (msg->getType())
 {
    case selectMSG:
         data->select_all_named(msg->getString());
         .....
```

The class **notifyMsg** contains the following data elmenents:

o Generic *source	Pointer to the object that sends the message
o Generic *destination	Pointer to the current recipient.
o MsgType type	Message type, see below
o Generic *object	Pointer to an object that must be derived from Generic and that can be destroyed when the message is going out of scope or is deleted.

o char *string	A string that can be destroyed when the message is going out of scope or is deleted.
o Boolean replied	Is set to FALSE until some object sets it to TRUE, signalling that a request was answered. Example: In the member function renewParameters of class mapper3D, a request is made to inform about the current viewpoint. An object of class ObserverPanel can answer this request and will set the variable in the notifyMsg to TRUE.

Message types. MsgType is an enum constant type and has currently the following possible settings (not all are used sofar):

enum MsgType { *unspecifiedMSG, triggerMSG, initializeMSG, resetMSG, updateMSG, repaintMSG, charstringMSG, objectMSG, selectMSG, unselectMSG, highlightMSG, listMSG, translateREQMSG, viewpointMSG, viewpointREQMSG* }

Public function members of the class **notifyMsg**

	notifyMsg()	Generator, nothing set.
	notifyMsg(Generic *src)	Generator, sender specified
	notifyMsg(Generic *src, MsgType t)	Generator, sender and type specified
	notifyMsg(Generic *src, char *txt, MsgType t=charstringMSG)	Generator, sender, a character string, and optionally a type specified. If the type is omitted, charstringMSG is assumed as default value
void	sendMsg(Generic *dest)	calles the routine receive_other in the destination dest. This member is overloaded, see FlowNode.h
void	send_to_all(UIObject *src)	This routine finds all objects that are connected with the sender src and sends them this message. This member is overloaded, see FlowNode.h
void	request_from_all(UIObject *src)	This routine finds all objects that are connected with the sender src and sends them this message. It stops finding objects when the request has been answered (some recipient has set the replied flag in the message). This member is overloaded, see FlowNode.h
void	exchange_string(char *new_str)	Deletes the old string in the message and replaces it by a copy of new_str. Relevant for translations of one string to another. Example: `DisplayG` returns communicates with a listobject to translate a structure name string into a color name string.
void	set_replied(), Boolean is_replied()	Used for communicating request notifyMsg's
Generic*	getSource()	returns the sender of the message

Generic*	getDestination()	returns the destination of the message
Generic*	getObject()	returns the object transported in the message
void	setObject(Generic *O)	overwrite old object pointer in the message. Relevant for exchanging an object on request. Example: In `ObserverPanel` the observer position is set via a special structure in which the information is contained, and which is known in both the `mapper3D` class and in the `ObserverPanel` class.
MsgType	getType()	returns the type of this message
char*	getString() const	returns a reference to the char string in the message, or a (char *) NULL.
notifyMsg()		Class Destructor. Attention: It deletes the char string if there is one, and it deletes the object in the notifyMsg if there is one.

3 Technical details of implementation

The following gives a short description of the particular computer environment that was used.

The AT&T C++ compiler release 2.1 was used. It is installed on the sparc station hsun23 at ATR which was in my use. The path to the compiler is currently:

`/export/hsun23/lang/CC`

The followin environmental variables have to be set in the Unix environment:

```
CCINC=/export/hsun23/lang/SC1.0/include/CC        ; includes for C++
GUIDEHOME=/usr/local/devguide                     ; to use devguide
UITHOME=/home/hsun23/wilhelms/uit/UIT             ; UIT library
GENERICHOME=/home/hsun23/wilhelms/uit/Generic     ; ...
UITSUPPL=/home/hsun23/wilhelms/uitlib             ; This library
LEDA=/home/hsun23/wilhelms/LEDA                   ; LEDA
HELPPATH=/usr/local/devguide/lib/locale:/usr/local/devguide/lib/help:/usr/local/openwin/lib/help
```

Currently the library is installed in the directory /homes/wilhelms/uitlib. The include files are in /homes/wilhelms/uit and the source files in /homes/wilhelms/uitlib/src. This documentation is in /homes/wilhelms/uitlib/doc.

The source directory /homes/wilhelms/uitlib/src contains a Makefile which can create the files in /homes/wilhelms/uit libUIC.so.2.0* and libUITSUPPL.a.

3.a The program observer

It can be found in the library `/homes/wilhelms/observer` There is a Makefile which, if the environmental variables are set as above, will compile and link observer. The main program is in observer_ui.cc, and observer_stubs.cc contains several handler functions. This program is not completed, even though it works properly. Additional functions are meant to be added soon: In the immediate future a method will be added to move, rotate and deform the displayed sketches of biological specimens in order to align serial sections under visual feed back. Further, a part for loading and saving files will be installed.

Right now, most of the buttons that are contained in the main panel have no real function, except that pushing them causes some empty handler functions to be called. The program observer was first constructed with `devguide` and then considerably modified, replacing for example the **ComponentDisplay** class object `MainCanvas` by a **mapper3D** class object `MainCanvas`.

3.b Support

I will continue the development of this library after returning to Columbus, Ohio. From time to time I will make updates available, via electronic mail or via ftp. I apprechiate any suggestions and bug reports. Have fun.

My E-mail address in Columbus Ohio is:

reiner@shs.ohio-state.edu

My address in Columbus:

Reiner Wilhelms, D. Sc.
Speech and Hearing Division
The Ohio State University
101 Pressey Hall, 1070 Carmack Rd
Columbus, OH 43210