XTANGO Algorithm Animation Designer's Package

John T. Stasko
Doug Hayes

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

stasko@cc.gatech.edu

October 21, 1992

This package is an implementation of the *path-transition* algorithm animation framework developed by John Stasko. (XTANGO = X-window Transition-based ANimation GeneratiOn.) It is designed to allow users to easily animate an algorithm or program without having to write any low-level graphics code. While the framework contains extensible models for images and transitions, in this package we have implemented ten types of images (lines, rectangles, circles, ellipses, polylines, polygons, splines, bitmaps, text, and composites) and ten types of transitions (fill, visibility, color, move, resize, zoom, raise, grab, lower, delete, refresh and delay). These primitives give us a compact working set for easy learning, yet they still provide sufficient depth to generate moderately complex animation sequences.

1 Data objects

The data types provided by this package are

- TANGO_LOC
  This is the handle that identifies a particular position in the viewing window. Locations are defined by real-valued (x, y) coordinate pairs.
- TANGO_IMAGE
  This is the handle that identifies a particular graphical image on the screen. In this implementation an image can be a line, rectangle, circle, ellipse, polyline, polygon, spline, bitmap, text, or a composite of the previous types.
- TANGO_PATH
  This is the handle that identifies an animation path along which an image is modified. A path can be thought of as a list of relative offsets, each relative to the previous position.
- TANGO_TRANS
  This is the handle that identifies an actual transition action. A transition action is comprised of 1)the actual type of the transition such as fill style change, visibility change, color change, motion, resizing, and so forth, or some combination thereof, 2)the image to which it applies, and 3)a path to "modify" along.

2 Entry Points

2.1 Algorithm Operations

TANGO contains one call that passes information from a program to its accompanying animation. This call should be placed at key points within the program being animated.

```
char *
TANGOalgoOp (name)
   char *name;
```

NOTE: This routine differs here in the varargs version than from its implementation in the non-varargs version.

This routine sends an algorithm operation named *name* to the animation controller. Only two possibilities for name exist, and you must make sure to call using both of them: one at the very beginning and one at the very end. The call TANGOalgoOp("BEGIN") should always be the first statement of your driver program and the call TANGOalgoOp("END") should always be the last.

2.2 Initialization/Termination

The animation package creates its own graphical display environment on top of the X11 Window system.

Animation window graphics are laid out in an infinite coordinate system with positive x values moving right and positive y values moving down. To start, the upper left corner is (0.0,0.0) and the lower right corner is (1.0,1.0). Locations outside the predefined range can be accessed; images displayed there simply will not be seen until those coordinates are scrolled into the viewing area.

```
void
TANGOinit ( )
```

This routine is used to initialize the TANGO package. It will be called automatically by TANGOalgoOp(mappingstruct, "BEGIN").

```
void
TANGOend ( )
```

This routine is used to terminate the TANGO package. It will be called automatically by TANGOalgoOp(mappingstruct, "END").

2.3 Locations

The ability to save various logical coordinate points in the viewing window is a powerful tool. Because locations are identified within a real-valued coordinate system, the same animation can be performed in windows of varying physical size without changing any of the code controlling the animation.

```
TANGO_LOC
TANGOloc_create (x,y)
   double x,y;
```

This routine creates a logical location with the given x and y coordinates.

```
double
TANGOloc_X (loc)
   TANGO_LOC loc;
```

This routine returns the x coordinate of the given TANGO_LOC.

```
double
TANGOloc_Y (loc)
   TANGO_LOC loc;
```

This routine returns the y coordinate of the given TANGO_LOC.

```
void
TANGOloc_inquire (loc,x,y)
   TANGO_LOC loc;
   double *x,*y;
```

This routine returns, through its parameters, the x and y values represented by the given TANGO_LOC.

```
TANGO_LOC
TANGOloc_modify (loc,x,y)
   TANGO_LOC loc;
   double x,y;
```

This routine creates a new TANGO_LOC that has coordinates equal to the given location plus the parameter values provided as arguments. (The values are simply added together).

```
int
TANGOloc_equal (loc1,loc2)
   TANGO_LOC loc1,loc2;
```

This routine returns 1 if the two locations have the same x and y coordinates (within a very small error factor) and 0 otherwise.

2.4 Images

Nine simple graphical objects (lines, rectangles, circles, ellipses, polygons, polylines, splines, bitmaps, and text) are the basis of the animation package. Rectangles, circles, ellipses, and polygons can have varying fill styles. Lines can have different thicknesses and styles. Currently, this base set is limited to provide a simpler basis to design animations.

```
TANGO_IMAGE
TANGOimage_create (type,p1,p2,...,pn)
   TANGO_IMAGE_TYPE type;
   param p1,p2,...,pn;
```

This routine allocates a graphical image and returns a handle to it. **Important:** The image will not become visible until the first frame of a subsequent transition occurs, however. The image *type* can be one of the following types: TANGO_IMAGE_TYPE_LINE, TANGO_IMAGE_TYPE_RECTANGLE, TANGO_IMAGE_TYPE_CIRCLE, TANGO_IMAGE_TYPE_ELLIPSE, TANGO_IMAGE_TYPE_POLYLINE, TANGO_IMAGE_TYPE_POLYGON, TANGO_IMAGE_TYPE_SPLINE, TANGO_IMAGE_TYPE_TEXT, TANGO_IMAGE_TYPE_BITMAP, or TANGO_IMAGE_TYPE_COMPOSITE. Depending upon the image's type, the subsequent parameters must follow the appropriate protocol (TC stands for TANGO_COLOR below):

LINE - (lx,ly,vis,color,sx,sy,wid,sty,arrow) double lx,ly,sx,sy,wid,sty; TC color; int vis,arrow;

The *lx* and *ly* parameters correspond to locations, and the *sx* and *sy* parameters correspond to sizes. Line sizes can be positive or negative. *wid* defines the width of the line; it can range from 0.0 to 1.0 (corresponding roughly to percentages). 0.0 corresponds to a thin line, and 1.0 corresponds to the thickest line. Currently, we have implemented three thicknesses that are achieved by values in the range 0.0 to 0.333, 0.333 to 0.667, and 0.667 to 1.0, respectively. *sty* defines the line's style. It ranges from 0.0 to 1.0 also. We have implemented three styles: 0.0 to 0.333 defines a dotted line, 0.333 to 0.667 defines a dashed line, and 0.667 to 1.0 defines a solid line. *color* gives the color of the image. This can be one of TANGO_COLOR_WHITE, TANGO_COLOR_BLACK, TANGO_COLOR_RED, TANGO_COLOR_ORANGE,

TANGO_COLOR_YELLOW, TANGO_COLOR_GREEN, TANGO_COLOR_BLUE, or TANGO_COLOR_MAROON, or it can be a TANGO_COLOR created via the call TANGOload_color. If the *vis* parameter is 1, the image is initially visible. If *vis* is 0, the image is initially invisible. *arrow* designates an arrow style: 0 means no arrows, 1 means a forward arrow, 2 means a backward arrow, and 3 means a bi-directional arrow.

RECTANGLE - (lx,ly,vis,color,sx,sy,fill) double lx,ly,sx,sy,fill; int vis; TC color;
Rectangles are located by their upper left corner, with only positive sizes ranging down and to the right allowed. The *fill* parameter is a value between 0.0 and 1.0. 0.0 corresponds to an unfilled outline and 1.0 corresponds to 100 per cent fill in the given color. (Forty graduated fill styles are actually implemented.)

CIRCLE - (lx,ly,vis,color,radius,fill) double lx,ly,radius,fill; int vis; TC color;
For circles, the location pair denotes the center of the circle, with the *radius* parameter defining the radius of the circle.

ELLIPSE - (lx,ly,vis,color,rx,ry,fill) double lx,ly,rx,ry,fill; int vis; TC color;
For ellipses, the location pair identifies the ellipse's center. The *rx* and *ry* values denote the ellipse's radii in x and y repspectively.

POLYLINE - (lx,ly,vis,clr,vert,vx,vy,wid,sty,arw; double lx,ly,vx[],vy[],wid,sty; int vis,vert,arw; TC clr;
The location pair identifies the beginning vertex of the polyline. The value *vert* identifies the number of vertices on the polyline. There can be a maximum of 8. The arrays *vx* and *vy* define the x and y **relative** offsets of the other polyline vertices from the location vertex given earlier, not from each other. The vertex identified by the *lx, ly* pair should not be included in these arrays, i.e., the arrays can be at most 7 elements. *wid* and *sty* define the width and style of the polyline segments just as in the line image type. *arrow* works just as in a line. In a polyline, the arrow's forward direction is determined by direction of the last polyline edge that is non-negative (vice-versa for backward).

POLYGON - (lx,ly,vis,color,sides,vx,vy,fill; double lx,ly,vx[],vy[],fill; int vis,sides; TC color;
The location pair identifies a vertex of the polygon. The value *sides* identifies the number of sides (or equivalently vertices) on the polygon. There can be a maximum of 8. The arrays *vx* and *vy* define the x and y **relative** offsets of the other polygon vertices from the location vertex given earlier. The vertex identified by the *lx, ly* pair should not be included in these arrays, i.e., the arrays can be at most 7 elements. Polygons can be concave or convex, but they should not be self-intersecting. *fill* gives the fill value of the polygon.

SPLINE - (lx,ly,vis,color,vert,vx,vy,wid,sty; double lx,ly,vx[],vy[],wid,sty; int vis,vert; TC color
The location pair identifies the beginning control point of the spline. The value *vert* identifies the number of points on the spline. There can be a maximum of 8. The arrays *vx* and *vy* define the x and y **relative** offsets of the other spline points from the location point given earlier. The point identified by the *lx, ly* pair should not be included in these arrays, i.e., the arrays can be at most 7 elements. *wid* and *sty* define the width and style of the spline just as in the line image type.

BITMAP - (lx,ly,vis,bmaps,num,wid,hei) double lx,ly; int vis,nu,wid,hei; int *bmaps;
A bitmap image is really a series of 2-d bitmaps. The location pair defines the top-left position of the image. *bmaps* is a 3-d integer array in which the first dimension is the number of bitmaps, second is the width of each bitmap, and third is the height of each bitmap. Each position in the array specifies a pixel color. The integers placed there correspond to the integer values for TANGO_COLORs. For example, the bitmap definition below shows a yellow line swing down across a black background. A bitmap begins showing the first 2-d bitmap array defined for it. (The SHUFFLE transition is useful to then cycle through the others.)

```
static int bmaps[3][4][4]= {
         {{1,1,1,1},
          {7,7,7,7},
          {7,7,7,7},
          {7,7,7,7}},

         {{7,7,7,1},
          {7,7,1,7},
          {7,1,7,7},
          {1,7,7,7}},
```

```
{{7,7,7,1},
 {7,7,7,1},
 {7,7,7,1},
 {7,7,7,1}}};
```

TEXT - (lx,ly,vis,color,fname,str,ctr) double lx,ly; TC color; int vis,ctr; char *str, *fname;
The location pair identifies the lower left corner of where the text will be placed if the *ctr* parameter is 0, or the center location of where the text will be placed if the parameter is 1. The *fname* parameter tells what font should be used (if NULL, the default font is used), and the *str* parameter gives the text to be displayed.

COMPOSITE - (lx,ly,vis,im) double lx,ly; int vis; TANGO_IMAGE_COMPONENT im[];
This option allows a composite image to be loaded. A composite image is one that is made up of primary subimages (all above) and that has its own location and visibility. The *im* argument should be an array of TANGO_IMAGE_COMPONENT objects. Each of these has the form

```
typedef struct _IMAGE_COMP {
      TANGO_IMAGE_TYPE type;
      char *args;
} TANGO_IMAGE_COMPONENT;
```

type is the type of the subimage. The *args* parameter is a string of values encoded as follows (note, no commas):

For line: "lx ly color sx sy wid sty arrow"
For rectangle: "lx ly color sx sy fill"
For circle: "lx ly color rad fill"
For ellipse: "lx lx color radx rady fill"
For polyline: "lx ly color wid sty arrow vertices vx1 vy1 vx2 vy2 ..."
For polygon: "lx ly color fill sides vx1 vy1 vx2 vy2 ..."
For spline: "lx ly color wid sty vertices vx1 vy1 vx2 vy2 ..."
For text: "lx ly color fontname str centered"

Each line defines a subimage in the composite image. The lx and ly values are *relative* locations with respect to the location given in the TANGOimage_create call. The color should be a color written out. The other parameters are of the same type and are used just like the above parameters for the basic image types. Note that bitmap subimages are not allowed in composites. The following example will define a door.

```
static TANGO_IMAGE_COMPONENT door[5] = {
   { TANGO_IMAGE_TYPE_RECTANGLE, "0.0 0.0 TANGO_COLOR_BLACK 0.1 0.2 0.0"},
   { TANGO_IMAGE_TYPE_RECTANGLE, "0.01 0.01 TANGO_COLOR_BLACK 0.08 0.08 0.0"},
   { TANGO_IMAGE_TYPE_RECTANGLE, "0.01 0.11 TANGO_COLOR_BLACK 0.08 0.08 0.0"},
   { TANGO_IMAGE_TYPE_CIRCLE, "0.09 0.1 TANGO_COLOR_BLACK 0.005 0.0"},
   { TANGO_IMAGE_TYPE_COMPOSITE, NULL}
};
```

Note that the array definition should always end with a COMPOSITE type that tells TANGO that this is the end of the definition.

Composite images are useful when a certain group of images (such as that of a stick person) is to be created and used multiple times in an animation. When an inquiry is made about the location of a composite image, the current location of the entire composite image is returned. Currently, transitions applied to a composite image will affect all the subimages in the same way. Ideally, this shouldn't happen, i.e., when we resize a person, we want the relative positions of the components to remain constant.

```
TANGO_IMAGE
TANGOimage_copy (image)
   TANGO_IMAGE image;
```

This routine returns a new TANGO_IMAGE which is an exact copy of the one given as a parameter (at that instant in time). Just like TANGOimage_create, the new image will not appear until a transition is performed.

```
TANGO_LOC
TANGOimage_loc (image,part)
   TANGO_IMAGE image;
   TANGO_PART_TYPE part;
```

This routine returns an TANGO_LOC that corresponds to the location of the given part of the given *image.* Valid TANGO_PART_TYPES include TANGO_PART_TYPE_C (center), and the compass directions TANGO_PART_TYPE_NW, TANGO_PART_TYPE_N, TANGO_PART_TYPE_NE, TANGO_PART_TYPE_E, TANGO_PART_TYPE_SE, TANGO_PART_TYPE_S, TANGO_PART_TYPE_SW, and TANGO_PART_TYPE_W. For rectangles and circles, actual locations on the image boundary (other than the center) are returned. For lines, ellipses, polylines, polygons, splines, and text, a bounding box location is returned. For composites, a bounding box of all the subimages is returned.

2.5 Paths

A path is the graphical basis upon which image modification is based. It can be thought of as a list of relative (x and y) offsets from the previous location. Under the move transition, the control points or offsets in the path define where the image should next be located. Think of a control point in a path like a frame in a motion picture. In subsequent frames, images have changed location by some small increment. If we iterate the frames in this series, the image appears to move along this path. The *length* of a path is the number of offsets it contains.

Paths are not only used for movement, however. Every transition utilizes a path argument to define its action.

```
TANGO_PATH
TANGOpath_create (n,xoffsets,yoffsets)
   int n;
   double xoffsets[],yoffsets[];
```

This routine creates a path that has length n, designated by the given sequence of x and y offset pairs.

```
TANGO_PATH
TANGOpath_load (file)
   char *file;
```

This routine loads a path from a file in which it has been stored (via TANGOpath_store). Consequently, this routine provides the capability to reuse paths from session to session. If the path was unable to be loaded, the routine returns NULL.

```
TANGO_PATH
TANGOpath_store (path,file)
   TANGO_PATH path;
   char *file;
```

This routine stores a path into a file for later use. The path may have been created in any valid way. If the path is stored successfully, the routine returns the same TANGO_PATH handle. If the store fails, the routine returns NULL.

```
int
TANGOpath_length (path)
   TANGO_PATH path;
```

This routine returns the numbers of offsets in the given path.

```
double
TANGOpath_dx (path)
   TANGO_PATH path;

double
TANGOpath_dy (path)
   TANGO_PATH path;
```

These routines return the x or y distance, respectively, that the given path traverses from start to finish.

```
TANGO_PATH
TANGOpath_type (type)
   TANGO_PATH_TYPE type;
```

This routine creates a path which is one of three basic path types. They are as follows:

TANGO_PATH_TYPE_STRAIGHT	move right
TANGO_PATH_TYPE_CLOCKWISE	move clockwise right in an upward arc
TANGO_PATH_TYPE_COUNTERCLOCKWISE	move counterclockwise left in an upward arc

Each of these paths when created will have 20 relative offset points and will change in x by a total value of 0.2 animation coordinate system units.

The paths created by this routine will probably not be utilized as is, although there is no reason why they cannot be. They are provided primarily as starting paths for use in the subsequent path modification routines.

```
TANGO_PATH
TANGOpath_color (color)
   TANGO_COLOR color;
```

This routine returns a one offset path that will change an image to the given color when the path is used as an argument to an TANGO_TRANS_TYPE_COLOR transition. The direction of the offset in the path determines the color to utilize. For example, if there is a black rectangle that you wish to be red, first get a path by calling TANGOpath_color(TANGO_COLOR_RED), then use the returned path as the third argument to TANGOtrans_create(TANGO_TRANS_TYPE_COLOR,image,path). For available colors, see TANGOimage_create. Note that colors created with TANGOload_color cannot be used as parameters to this function.

```
TANGO_PATH
TANGOpath_add_tail (path,num)
   TANGO_PATH path;
   int num;
```

```
TANGO_PATH
TANGOpath_add_head (path,num)
   TANGO_PATH path;
   int num;
```

These routines provide ways of elongating a path. The parameter *num* designates the number of null ($\langle 0.0, 0.0 \rangle$) offsets that will be apppended to the given *path* at either the head or tail in order to produce a new path that is returned.

```
TANGO_PATH
TANGOpath_delete_tail (path,num)
   TANGO_PATH path;
   int num;

TANGO_PATH
TANGOpath_delete_head (path,num)
   TANGO_PATH path;
   int num;
```

These routines provide ways of shortening a path. The parameter *num* designates the number of offsets that will be removed from the head or tail of the given path in order to produce a new path that is returned.

```
TANGO_PATH
TANGOpath_reverse (path)
   TANGO_PATH path;
```

This routine returns a path that is the inverse of the given path. Being an inverse path means that the order of offsets is reversed, and each offset points in the exact opposite direction. This routine provides a way for an image to retrace its tracks from a movement transition.

```
TANGO_PATH
TANGOpath_rotate (path,deg)
   TANGO_PATH path;
   int deg;
```

This routine returns a path that corresponds to the rotation of the given path in a counter-clockwise motion. The number of degrees to rotate is provided by the integer parameter *deg* which should be between 0 and 360.

```
TANGO_PATH
TANGOpath_interpolate (path,factor)
   TANGO_PATH path;
   double factor;
```

This routine returns a path in which the number of offsets is modified by a factor given by the *factor* parameter. If a path with length 10 is interpolated with a factor of 2.0, the returned path will have 20 offsets. If the factor is 0.3, the path will have 3 offsets. Straight linear interpolation is used. Consequently, when interpolating a path with choppy, wavy motion characteristics, individual extrema may be lost with certain *factor* parameters.

```
TANGO_PATH
TANGOpath_scale (path,dx,dy)
   TANGO_PATH path;
   double dx, dy;
```

This routine returns a path in which each offset is scaled by the given factors in x and y. That is, each offsets' x and y values are mulitplied by the dx and dy factors, respectively. So, for example, to return a path that would be the reflection of a path across an imaginary vertical line, choose x = -1.0 and y = 1.0.

```
TANGO_PATH
TANGOpath_extend (path,dx,dy)
   TANGO_PATH path;
   double dx, dy;
```

This routine returns a path in which each offset is extended by the given factors in x and y. That is, each offsets' x and y values have the dx and dy factors added to them, respectively.

```
TANGO_PATH
TANGOpath_copy (path)
   TANGO_PATH path;
```

This routine returns a new path that has an exact duplicate list of offsets as the given path.

```
TANGO_PATH
TANGOpath_null (num)
   int num;
```

This routine provides a way of creating a null path (all $\langle 0.0,0.0 \rangle$ relative offsets that "go nowhere") that is the given number of offsets long. This routine is useful for generating paths in which only the number of offsets is necessary, not their particular values.

```
TANGO_PATH
TANGOpath_iterate (path,factor)
   TANGO_PATH path;
   double factor;
```

This routine returns a path which is an iteration of a given path. The *factor* parameter provides how many times the given path should be repeated in the path to be created.

```
TANGO_PATH
TANGOpath_concatenate (num,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10)
   int num;
   TANGO_PATH p1,p2,p3,p4,p5,p6,p7,p8,p9,p10;
```

This routine creates a path which is the concatenation of some number of other paths. The *num* parameter provides how many paths are to be concatenated, and the pi parameters contain the paths themselves. The paths will be concatenated in order from the 1 position on to the num position. The first offset of a path is relative to the last offset of the previous path.

```
TANGO_PATH
TANGOpath_compose (num,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10)
   int num;
   TANGO_PATH p1,p2,p3,p4,p5,p6,p7,p8,p9,p10;
```

This routine returns a path which is the composition of the *num* paths given in the *p1* to *p*num parameters. By composition, we mean a cumulative combination on an offset by offset basis. If a path is thought of as a vector, the composition of paths produces a new vector that has the same length as the originals and is like the vector sum of the original path vectors. Only paths with an equal number of offsets can be composed. Otherwise, the routine returns NULL.

Essentially, path composition takes all the first relative $\langle x,y \rangle$ offsets and adds them together to make a new cumulative first $\langle x,y \rangle$ offset. This is done for each offset in the paths.

```
TANGO_PATH
TANGOpath_distance (fromloc,toloc,distance)
   TANGO_LOC fromloc;
   TANGO_LOC toloc;
   double distance;
```

This routine returns a path that proceeds in a straight line from the given *fromloc* to the given *toloc* with the condition that the path will have an offset every time the given *distance* has been covered. If the given distance is larger than the distance between the fromloc and toloc, the path is only given one offset. This routine is useful to create paths that will make images move at the same velocity.

```
TANGO_PATH
TANGOpath_example (fromloc,toloc,path)
   TANGO_LOC fromloc;
   TANGO_LOC toloc;
   TANGO_PATH path;
```

This routine returns a path which "looks like" the given example path, but which would move from the logical *fromloc* to the logical *toloc*. The created path will have the same number of offsets as the given path. The general flow of movement in the example path is followed as closely as possible by maintaining the same ratios between control points in both paths. Clearly, however, if the two paths move in opposite directions, they will not look much alike. Typically, this routine will be used when one wants an image to end up in some specific position, with the image following some rough, given trajectory path in order to get there. If the *path* argument is null, a straight line path of 20 equally spaced offsets is used as the example path.

```
TANGO_PATH
TANGOpath_motion (fromloc,toloc,type)
   TANGO_LOC fromloc;
   TANGO_LOC toloc;
   TANGO_PATH_TYPE type;
```

This routine returns a path with movement characteristics of the given *type*, but that also begins at the location with logical x and y coordinates of *fromloc* and proceeds to the location with logical x and y coordinates of the *toloc*. Note that the provided from and to locations need not have absolute screen coordinate meaning. Because a path is made up of a group of relative offsets, these locations are provided just as necessary aids in the path creation. For example, the same path will be created using from = $\langle 0.2,0.2 \rangle$ and to = $\langle 0.4,0.4 \rangle$ as using from = $\langle 0.7,0.7 \rangle$ and to = $\langle 0.9,0.9 \rangle$. Most often however, the provided locations will be specific window coordinates chosen to move a certain image to a specific location.

The created path will contain 20 offsets and will be modelled to fit the given path type, which may be one of TANGO_PATH_TYPE_STRAIGHT, TANGO_PATH_TYPE_CLOCKWISE, or TANGO_PATH_TYPE_COUNTERCLOCKWISE. The straight option creates a path that is a direct line from *fromloc* to *toloc* with equally spaced offsets. The two clock motions will create a path that curves in the given clock motion direction.

Essentially, this routine is just TANGOpath_example(fromloc, toloc, TANGOpath_type(type)).

```
TANGO_PATH
TANGOpath_smooth (path)
   TANGO_PATH path;
```

This routine returns a path which is a “smoother” version of the given path. Essentially, each offsets’ value is determined by averaging in the neighboring offsets’ values. Currently, we use 2 times the designated offset plus one time the previous and subsequent offsets.

```
void
TANGOpath_free (num,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10)
   int num;
   TANGO_PATH p1,p2,p3,p4,p5,p6,p7,p8,p9,p10;
```

This routine frees the memory space occupied by an TANGO_PATH. Any number from one to ten paths can be passed in, (the *num* parameter indicates how many), and each of the path’s memory space will be freed. It is a good idea to free the space once a path will no longer be used.

2.6 Transitions

Transitions are the actual animation actions which affect images on the screen. Ten basic types have been implemented in this package: change in visibility, change in fill style (shade), change in color, movement, resizing, raising, lowering, grab, delete, refresh, and the delay action. These seven basic types can be combined via the complex transition operators iteration, concatenation, and composition. A transition is made up of a series or list of frames. Modifications to images occur within each frame.

All of the following transition routines which return an TANGO do not actually perform the transition. They merely create a logical object which can later be performed (repeatedly) in order to carry out the desired action. Transitions are fully recursive in that compose transitions can be concatenated, iterate transitions can be composed with “single” transitions, etc.

```
TANGO_TRANS
TANGOtrans_create (type,image,path)
   TANGO_TRANS_TYPE type;
   TANGO_IMAGE image;
   TANGO_PATH path;
```

This is the routine that creates a transition of one of the basic types. The *type* parameter should be one of TANGO_TRANS_TYPE_MOVE, TANGO_TRANS_TYPE_VISIBLE, TANGO_TRANS_TYPE_FILL, TANGO_TRANS_TYPE_RESIZE, TANGO_TRANS_TYPE_RESIZE1, TANGO_TRANS_TYPE_RESIZE2, TANGO_TRANS_TYPE_RESIZE3, TANGO_TRANS_TYPE_RESIZE4, TANGO_TRANS_TYPE_RESIZE5, TANGO_TRANS_TYPE_RESIZE6, TANGO_TRANS_TYPE_RESIZE7, TANGO_TRANS_TYPE_GRAB1, TANGO_TRANS_TYPE_GRAB2, TANGO_TRANS_TYPE_GRAB3, TANGO_TRANS_TYPE_GRAB4, TANGO_TRANS_TYPE_GRAB5, TANGO_TRANS_TYPE_GRAB6, TANGO_TRANS_TYPE_GRAB7, TANGO_TRANS_TYPE_SHUFFLE, TANGO_TRANS_TYPE_RAISE, TANGO_TRANS_TYPE_LOWER, TANGO_TRANS_TYPE_DELETE, TANGO_TRANS_TYPE_ZOOM, TANGO_TRANS_REFRESH, or TANGO_TRANS_TYPE_DELAY. Individual transitions operate as follows:

MOVE - move the given *image* along the given *path*. The first movement of the image corresponds to the first relative offset in the path. All these relative offsets are with respect to the image’s previous position on the screen.

VISIBLE - switch the visibility of the given *image* for each offset in the given *path*. At each offset in the path, if the image is visible, it will become invisible, and vice-versa.

COLOR - change the given *image* to the color indicated by the given *path*. See the routine TANGOpath_color for how to create a path that will change an image to a certain color.

FILL - change the ”fill” component of the given image. This works differently for different types of images. For rectangles, circles, ellipses, and polygons, this transition alters the image’s fill style value by the

x value of the offsets in the path. The x value is simply added in. If the image's fill value goes below 0.0, it is automatically set back to 0.0. If it goes above 1.0, it is set back to 1.0. Actually, the full range of values between 0.0 and 1.0 is not available. We currently implement 40 different fills that range between the two extremes. For lines, polylines, and splines, the x value of each offset is added to the line's *width* parameter value, and the y value is added to the line's style parameter value (see TANGOimage_create for information on width and styles). Extreme values are reset to 0.0 and 1.0 as in rectangles and circles.

RESIZE - resize the given *image* along the given *path*. The various types of images each have a "method" in which they are resized. Since lines can have positive or negative sizes, they are resized by altering the line's size for each offset in the path. Rectangles can have only positive sizes, so resizing a rectangle corresponds to "dragging" the lower right corner of the rectangle along the given path. If one of the rectangle's dimensions would become negative, it is set to 0. Circles are resized by modifying the circle's radius by the amount given in the x component of each offset in the path. On an ellipse, the resize transition adds the x value to the ellipse's x radius and the y value to the y radius. For polylines, polygons, and splines the transitions RESIZE1-RESIZE7 modify relative positions of the respective vertex number, plus all others after it in numerical order, by the relative offsets of the path. These transitions are useful, for example, with a forward arrow polyline that has many of its edges compressed down to start. They can subsequently be grown out in all different directions, one at a time. Currently, resizing has no affect on text.

GRAB - The GRAB transitions modify polylines, polygons, and splines by altering the relative position of a particular vertex on the image (except the one denoting the image's position) by the relative offsets of the path. As opposed to RESIZE, the transitions GRAB1-GRAB7 alter only one particular vertex in the image's definition. Think of grabbing that vertex and swinging it around while all the other points stay anchored. GRABs have no effect on other images.

SHUFFLE - This transition type only affects BITMAP image type. For each offset in the path argument, the next 2-d bitmap in the image's series will be displayed. The series is considered circular, i.e., the successor of the last bitmap is the first bitmap. This transition is useful to get a certain effect such as a door opening. To do this, define a BITMAP image with a series of individual bitmaps showing the various stages of the door opening. Then run a SHUFFLE transition to actually show this happen.

RAISE - bring the given *image* to the viewing plane closest to the viewer. The image's position is not changed, only its relative ordering (top to bottom) with respect to other images.

LOWER - push the given *image* to the viewing plane farthest from the viewer. The image's position is not changed, only its relative ordering (top to bottom) with respect to other images. It will possibly be obscured by every other image.

ZOOM - have the window zoom in or out on the positional location of the image argument. The x and y components of the path determine how the zooming should occur. They should be between 0.0 and 1.0. Values close to 1.0 will zoom in very tight. Values near 0.0 do a slow zoom. To get a nice slow, smooth zoom in on an object, use a path with quite a few offsets all of equal "small" offset values, such as $\langle 0.1, 0.1 \rangle$. Positive offset values zoom the display in, negative offset values pull the display back out.

DELETE - make an image invisible (if not already) and remove it from any possible later usage. Any attempts to use this image as a parameter in subsequent TANGO calls will result in error messages. Also, pre-existing transitions with this image as a parameter should not be performed.

REFRESH - redraw the entire animation window, restoring it to a pristine state. This is useful when running TANGO in fast mode (non-total frame by frame refresh) and you would like to clean up damage to the screen. A refresh is done for each offset in the path, i.e., there's no need for more than one offset in the path. The image used in this transition does not matter and can be NULL.

DELAY - create a transition which has a null frame for each offset in the path. By null frame, we mean that no action or modification is performed. This transition is helpful when combined with concatenation in order to produce actions that commence at varying times. Note that NULL is an acceptable image for use in the delay transition.

```
TANGO_TRANS
TANGOtrans_iterate (trans,num)
   TANGO_TRANS trans;
   int num;
```

This routine returns a transition which corresponds to performing the given *trans num* times, one immediately after the other. One example use of this routine is to make an image repeat a given behavior.

```
TANGO_TRANS
TANGOtrans_concatenate (num,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10)
   int num;
   TANGO_TRANS t1,t2,t3,t4,t5,t6,t7,t8,t9,t10;
```

This routine returns a transition which corresponds to the concatenation of the transitions in the *ti* parameters. The parameter *num* denotes the number of valid transitions being passed in. The order of the incoming transitions corresponds to the order that they will be performed in the concatenation with transition *t1* being first. This one logical transition can then be composed, iterated, etc., with other transitions to achieve various desired animations.

```
TANGO_TRANS
TANGOtrans_compose (num,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10)
   int num;
   TANGO_TRANS t1,t2,t3,t4,t5,t6,t7,t8,t9,t10;
```

This routine returns a transition which is the composition of the *num* transitions given in the subsequent parameters. To call the function and compose 2 transitions, call TANGOtrans_compose(2,t1,t2). To compose more than 10 transitions, you need to call this routine twice with the result of the first call as a parameter to the second call. By composition, we mean the "concurrent" execution of all the transitions involved. When transitions are composed, individual frames of the transition are combined, e.g., the first frames, the second frames, etc. For transitions, each offset in the path utilized corresponds to one frame. Transitions of unequal "length" can be composed. Transitions shorter than the longest transition will simply have null action frames added to their tails. Consequently, all the action will start together but will finish according to how many frames are in the transition.

This routine is very important because it provides the ability to have many images moving and changing around the screen at the same time.

```
void
TANGOtrans_perform (trans)
   TANGO_TRANS trans;
```

This routine actually carries out the given transition which has been created via one of the transition creation routines. All the involved graphical images are animated in the ways specified in the transition when it was created.

```
void
TANGOtrans_free (num,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10)
   int num;
   TANGO_TRANS t1,t2,t3,t4,t5,t6,t7,t8,t9,t10;
```

This routine frees the space associated with TANGO_TRANSs. *num* indicates the number of transitions that are being passed in through the first *t1* to *tnum* parameters. Any number from one to ten is allowable. The space associated with each of the transitions will be freed. It is a good idea to call this routine when a transition will no longer be used.

2.7 Miscellaneous

TANGO provides limited input capabilities. Currently, it allows the end-user to interactively select real-value coordinates and image objects in the animation window. Data can then be passed back to the driving program through return values of animation scenes.

TANGO also provides a handle onto all of the valid X colors. There is a routine that can be used to load an X color in TANGO's color database.

TANGO also provides ways for a designer to zoom and/or pan the display. There is a routine to set the window boundary coordinates (also one to inquire the current coordinates).

```
int
TANGOinput_coord (x,y)
   double *x,*y;
```

This routine prompts the user to select a coordinate from the animation window. If a successful choice is made, the routine returns 1 and places the x and y values into the respective parameters. If for some reason the selection was not successful (e.g., the mouse selection was made outside the tango animation window), the routine returns 0 and places 0.0 in both parameters.

```
int
TANGOinput_image (image)
   TANGO_IMAGE *image;
```

This routine prompts the user to select an image from the animation window. The user clicks down the mouse and this routine returns the uppermost **visible** image object whose bounding box (with a little padding, hence you can select vertical and horizontal lines) encloses the point of the mouse selection. If a successful image choice is made, the routine returns 1 and places the chosen TANGO_IMAGE into the *image* parameter. If no image was selected, the routine returns 0 and places NULL in the parameter. Be careful with lines; they often have big bounding boxes. The raise and lower transitions are often useful aids for inputting images.

```
TANGO_COLOR
TANGOload_color (colorname)
   char *colorname;
```

This routine loads a color into TANGO's database of colors. The color string name should be one from X's color name database (usually in /usr/lib/X11/rgb.txt on most systems). The returned TANGO_COLOR can then be used in subsequent image creation calls. Note that the color cannot be used as a parameter to TANGOpath_color. Only one of the basic 8 colors can be used there.

```
int
TANGOset_bgcolor (colorname)
   char *colorname;
```

This routine can be used to change the background color of the xtango display area. Its effect is immediate. The default is white.


```
int
TANGOset_bgcolor (colorname)
   char *colorname;
```

This routine can be used to change the background color of the xtango display area. Its effect is immediate. The default is white.

```
int
TANGOset_coord (lx,by,rx,ty)
   double lx, by, rx, ty;
```

This routine is used to manually set the window boundary coordinates which start by default at 0.0 to 1.0 in both directions (origin is at upper left corner). This routine then allows you to zoom or pan the display from with an animation file. It is advisable to maintain the same aspect ratio between x and y or weird things may happen. A sequence of calls to this routine with gradually tightening window coordinates will provide a very nice smooth zooming functionality.

```
int
TANGOinq_coord (lx,by,rx,ty)
   double *lx, *by, *rx, *ty;
```

This routine returns the current value of the window's boundary coordinates.

2.8 TWIST Image and Location Grouping

These routines provide user-level, high-level, friendly access to the TANGO calls. They primarily consist of routines to create arrays or groups of images and locations. (TWIST = Tango's Wonderful Image Synthesis Tool). It is a form of higher-level macro facility for more convenient animation design and calls.

2.8.1 Locations

The routines provide the capabilities to create one-dimensional and two-dimensional arrays of locations, as well as a binary tree of locations.

```
void
TWISTcreate_loc_array (assoc,id,num,horiz,x,y,spacing)
   char *assoc;
   int id;
   int num;
   int horiz;
   double x,y;
   double spacing;
```

This routine creates a one-dimensional line of TANGO_LOCs. The locations can be accessed through associations. If *assoc* is non-null, the given string is used as the association name used to set up the locations. If it is null, the association is "ID". *id* is the identifier used as a parameter to the association. *num* provides the number of locations to create. If *horiz* is 1, the locations are set up in a horizontal row; if *horiz* is 0, a vertical column is created. x and y denote the starting location for the line, topmost location if vertical, and leftmost if horizontal. *spacing* gives the amount of space to place between locations.

Note: to reference individual locations after this routine has been called, use the call ASSOCretrieve("ID", id, index#) assuming that no association is given.

```
void
TWISTcreate_2d_loc_array (assoc,id,rows,cols,x,y,xspacing,yspacing)
   char *assoc;
   int id;
   int rows,cols;
   double x,y;
   double xspacing,yspacing;
```

This routine creates a two-dimensional grid of TANGO_LOCs. If *assoc* is NULL, the association "ID3" is utilized. If *assoc* is provided, it is used. *id* is the identifier used as the first parameter to the association set up for the locations. *rows* and *cols* indicate the number of rows and columns to be created. *x* and *y* denote the location of the upper left corner of the grid. *xspacing* and *yspacing* provide the amount of space to place between locations. .pp Note: to reference a particular location after the routine has been used, make the call ASSOCretrieve("ID3", id, row, col). They start at 0.

```
void
TWISTcreate_bintree (id,lx,by,rx,ty,edgelevels)
   int id;
   double lx,by,rx,ty;
   int edgelevels;
```

This routine lays out a binary tree within the bounding box designated by *lx, by, rx, ty*. The tree will contain the given number of edge levels (e.g., "1" designates a root and two children). All this routine does is to create a TANGO_LOC where each of the nodes should be placed (top-most point). It does NOT do any image creation. This routine also sets up associations that are helpful for accessing the TANGO_LOCs. Each location can be accessed through the ASSOCretrieve("NODE",id,i,j) call, where *id* is the parameter passed into this routine, *i* is the depth of the node (root is depth 0), and *j* is the ordering number of the node across its level. (Leftmost nodes start at position 0.) For example, *i*=2 and *j*=2 retrieves the location that is the left child of the right child of the root. Locations can also also be accessed by the call ASSOCretrieve("NUMBER",id,i) call, where *i* is the numeric placement of a node, in an order from top-to-bottom in depth, and then left-to-right across the tree. The root is node 0; its left child is 1 and its right child is 2, etc. This routine also sets up associations for accessing child and parent nodes. Calling ASSOCretrieve("PARENT",id,loc) returns the location that is the parent of *loc* (*loc* is a TANGO_LOC). The calls ASSOCretrieve("LCHILD",id,loc) and ASSOCretrieve("RCHILD",id,loc) return the left and right child locations for *loc* respectively.

This routine allows you to draw the style of node and edge that you like. It merely lays out the positions (top-center justified) where they should go. Note that by providing different *id*s, you can get multiple trees in the same animation window. Also note that the leaf node locations will reside on the horizontal line with *y*=*by*. Trying to retrieve the child of a leaf or the parent of the root returns NULL.

2.8.2 Images

These routines allow the creation of arrays or graphs of images.

```
void
TWISTcreate_image_array (as,id,n,ty,hrz,jst,x,y,xs,xqu,ys,yqu,sp,vis,col,fill,wid,sty)
   char *as;
   int id;
   int n;
   TANGO_IMAGE_TYPE ty;
   int hrz;
   int jst;
   double x,y;
   double xs[];
   double xqu;
   double ys[];
   double yqu;
   double sp;
   int vis;
   int col;
   double fill;
   double wid;
```

```
   double sty;
```

This routine creates a one-dimensional array (row or column) of images. The images can be accessed through associations. If *as* is NULL, the default association “ID” is utilized. An image can be “gotten” by its id and index number. If an *as* string is provided, it is used to make this identifying association. *id* identifies the images to the association package. *n* gives the number of images in the array. All the images are of the given TANGO_IMAGE_TYPE *ty*. This type can be TANGO_IMAGE_TYPE_LINE, TANGO_IMAGE_TYPE_RECTANGLE, or TANGO_IMAGE_TYPE_CIRCLE. If the parameter *hrz* is 1, a horizontal row of images is created. If the parameter is 0, a vertical column is created. *jst* provides the justification of the images. If a row is chosen, the images are bottom-justified when *jst* is 0, middle-justified if 1, and top-justified if 2. If a vertical column is chosen, the images are left-justified if *jst* is 0, middle-justified if 1, and right-justified if 2. *x* and *y* provide the starting location of the array of images. For a row, the *x* value denotes the left starting location, and the *y* value is the bottom, center, or top of the images depending on their justification. If a column is chosen, the *y* value denotes the top of the first image in the column, and the *x* value denotes the left, center, or right alignment position of the images. The *(x,y)* and *qu* parameters for x and y allow the images to be varying sizes. If the x's pointer parameter is NULL, all images are constant in the appropriate dimension with the given *qu* value denoting the size. To get varying heights of images, the *(x,y)s* parameter should point to an array (of length equal to *n*) of real numbers (ranging between 0.0 and 1.0). When the images are drawn, each image's height will be its indexed spot in *ys* times the *yqu* argument. Widths are determined by the *x* arguments. For circles, only the *x* parameters matter. They affect the circles' radii. *sp* provides the amount of space between images along the main axis direction. If *vis* is 0, the images are initially invisible. (1 makes them visible.) *col* indicates the color that the images will be. *fill* should be in the range 0.0 (outline) to 1.0 (solid color fill). *wid* and *sty* apply to lines. They should be in the range 0.0 to 1.0.

```
void
TWISTcreate_graph (assoc,id,num,loc,adj,im_type,size,fill)
   char *assoc;
   int id;
   int num;
   TANGO_LOC loc[];
   int adj[][50];
   TANGO_IMAGE_TYPE im_type;
   double size;
   double fill;
```

This routine draws the image of a graph given the locations of its vertices and its adjacency matrix. The graph is drawn in the classical manner with each vertex represented by a circular or square node, with connected vertices joined by a line. *id* and *num* identify the graph and the number of vertices. *loc* is an array of TANGO_LOCs specifying the locations of the vertices. *adj* is the standard adjacency matrix denoting which vertices are connected by an edge. 1 indicates connected, 0 indicates not connected. *im_type* gives the TANGO_IMAGE_TYPE to make each vertex's graphical image. This can be either TANGO_IMAGE_TYPE_RECTANGLE or TANGO_IMAGE_TYPE_CIRCLE. *size* gives the size of each image (side of the square for a rectangle, and radius for a circle.) *fill* applies to the image nodes, and should be a value in the range 0.0 to 1.0. (Graphs are always drawn in black color.) The following associations are set up by this routine: If *assoc* is null, under the association “ID” with parameters *id* and index, each TANGO_LOC can be accessed. (If an *assoc* is provided, it is used there.) Also, given the inquiry ASSOCretrieve(”IMAGE_AT”,*id*,tango_loc), the particular TANGO_IMAGES can be acquired. Finally, to access the TANGO_IMAGE which is the line between two vertices, make the call ASSOCretrieve(”EDGE_BETWEEN”,*id*,from_index,to_index). (The from and to ordering does not matter. Either way will retrieve the image.)

2.9 Associations

These routines provide the intermediary between the data of a program and the animation objects associated with this data. An association is defined by a name and the number of parameters to the association. The number of parameters can be from 0 to 5. The parameters or data object stored with an association can be any valid 32 bit quantity such as an integer or a pointer. We have indicated them as integers in the following routine definitions.

```
void
ASSOCinit ()
```

This routine initializes the ASSOC package and is called automatically by all the other ASSOC routines if the package has not already been initialized. It makes the reserved associations "ID" (2 parameters), "ID3" (3 parameters), and "IMAGE_AT" (2 parameters).

```
void
ASSOCmake (name,params)
   char *name;
   int params;
```

This routine creates an association for later use. Note that an association must be created with this routine before subsequent calls to ASSOCstore and ASSOCretrieve can be issued for the association. *name* provides the name of the association, and *params* provides the number of indexing parameters needed to reference a record of the association.

```
void
ASSOCstore (name,p1,p2,p3,p4,p5,p6)
   char *name;
   int p1,p2,p3,p4,p5,p6;
```

This routine allows a record to be stored under a particular association and set of indices. *name* gives the name of the association. The indexing parameters are sent in *p1* through *pn* assuming the association was made with *n* parameters. The record to be stored is in the *n+1* parameter. For instance, to store the record 29 under the association "ID" with indices 12334 and 2, use the call ASSOCstore("ID", 12334, 2, 29). Note that TANGO_IMAGEs, TANGO_LOCs, TANGO_PATHs, and TANGO_TRANSs are valid 32 bit quantities and they can be stored in associations.

```
int
ASSOCretrieve (name,p1,p2,p3,p4,p5)
   char *name;
   int p1,p2,p3,p4,p5;
```

This routine is used to retrieve the data record stored under a given association and set of indices. *name* provides the name of the association; the indices are given in the following parameters. To retrieve the record stored in the example above in ASSOCstore, make the call ASSOCretrieve("ID", 12334, 2). Typically, the value returned must be cast into its appropriate format such as a pointer to a character or an TANGO_IMAGE for example. Note that if no object has been stored under the provided association parameters, the routine returns NULL and prints out an error message (usually long and gory).

```
int
ASSOCmember (name,p1,p2,p3,p4,p5)
   char *name;
   int p1,p2,p3,p4,p5;
```

This routine is used to determine if a particular association and set of parameters has a data object stored under it. *name* provides the name of the association; the indices are given in the following parameters. The routine returns 1 if there is a data object stored, and 0 if no association store has been done. Basically, this routine provides a safe way to check for data object existence, in the case of uncertainty, before calling ASSOCretrieve.

```
int
ASSOCdelete (name,p1,p2,p3,p4,p5)
   char *name;
   int p1,p2,p3,p4,p5;
```

This routine is used to remove a data object from being stored under an association. Utilize the exact same parameters as you did to store the object. After this routine has been called, subsequent calls to ASSOCmember will return 0, and ASSOCretrieve returns NULL.

2.10 TAP Routines

These routines provide high-level user access to certain TANGO calls. They consist of routines designed to create transitions for certain often used actions. (TAP = TANGO's high-leve Animation Package). Almost all the routines return a transition, rather than actually performing it; in this way, the transition can be manipulated outside the corresponding TAP routine (i.e. the transitions can be concatenated, composed, etc., if needed). TAP, like TWIST, is a form of higher-level macro facility for easier and faster animation implementation; however, it is not absolutely necessary to use the routines, as any of the transitions could be created by the user.

```
void
TAPshow (image)
   TANGO_IMAGE image;
```

This routine simply displays *image* (i.e. uses a delay transition to display the object). Since an image will not appear until a subsequent animation occurs, this routine is useful for making sure that an image is properly displayed.

```
TANGO_TRANS
TAPfill (image)
   TANGO_IMAGE image;
```

This routine creates the transition needed to fill the given image to a solid color. The function then returns the transition.

```
TANGO_TRANS
TAPcolor (image, color)
   TANGO_IMAGE image;
   TANGO_COLOR color;
```

This routine creates the necessary transition to change *image* to the given color and returns this transition.

```
TANGO_TRANS
TAPvis_toggle(image)
   TANGO_IMAGE image;
```

This routine creates the transition required to toggle the visibility of *image* and returns this transition.

```
TANGO_TRANS
TAPjump(image, part, location)
   TANGO_IMAGE image;
   TANGO_PART_TYPE part;
   TANGO_LOC location;
```

This routine sets up a transition to move *image* to the given location using the specified part as a "handle." The transition uses a path of length one, so essentially, the image will appear to "jump" from its current location to the new location. The necessary transition is returned.

```
TANGO_TRANS
TAPmove(image, part, location)
   TANGO_IMAGE image;
   TANGO_PART_TYPE part;
   TANGO_LOC location;
```

This routine sets up a transition to move *image* from the location of its indicated *part* to the given location . This routine is similar to TAPjump; however, instead of using a path of length one to create the transition, it uses TANGOpath_motion to create the transition, which means that the path length is 20. The required transition is returned.

```
TANGO_TRANS
TAPtraverse(image, part, location, motion)
   TANGO_IMAGE image;
   TANGO_PART_TYPE part;
   TANGO_LOC location;
   TANGO_PATH_TYPE motion;
```

This routine creates the transition necessary to move an object (using the specified part of the object) from its current position to *location*. The type of path that the object follows is given by the parameter *motion*; *motion* is simply a path type, i.e. it is of the same form as the path types for the TANGOpath_motion routine.

```
TANGO_TRANS
TAPswitch(image1, image2)
   TANGO_IMAGE image1, image2;
```

This routine swaps two images; that is, *image1* will acquire *image2's* location and vice versa. The path that each image follows has a length of one, so the two images will appear to "jump" to their new locations.

```
TANGO_TRANS
TAPexchange(image1, image2)
   TANGO_IMAGE image1;
   TANGO_IMAGE image2;
```

This routine exchanges the position of two images. It is similar to TAPswitch; however, it moves each image over a path of length 20 using TANGOpath_motion.

```
TANGO_TRANS
TAPflash(image, num_flash)
   TANGO_IMAGE image;
   int num_flash;
```

This routine causes an object to toggle between visible and invisible for the given number of times. This makes an object appear to flash.