POLKA Animation Designer's Package

John T. Stasko

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

stasko@cc.gatech.edu

Release 1.25
December 19, 1997

This package implements a structured graphics and animation design toolkit in C++. With it, you can create color, smooth, 2 1/2-dimensional animations on top of the X11 Window System. It is particularly good for creating algorithm animations. The package has three levels of abstraction. The first and highest level is the Animator. You will need to create one Animator for each program to be animated. The Animator primarily handles program event reception. The second level is the View, which is a window onto the program. Multiple animation views can be open on one program. An Animator will drive all the views with which it is associated. Finally, there's the animation constituent level and the three main classes of objects contained therein: AnimObject, Location, and Action. Below we describe these levels in more detail.

Contents

1 Animator level

At the top level is the Animator class. Each program or algorithm animation should have at most one corresponding Animator object (actually a subclass of Animator will be created). The Animator receives the animation events from the program being viewed. It stores the event name and parameters in protected variables, and then calls a Controller virtual function. What you need to do is to derive a subclass of Animator and design your own controller function. The controller function needs to check what event just occurred and then call the appropriate animation view scenes.

Note that it is possible to build animations without using an Animator object. You can just have your code call View scenes directly.

Here is the Animator class:

```
const int MAXPARAMS = 16;

class Animator {
  protected:
    char    AlgoEvtName[32];
    int     AnimInts[MAXPARAMS];
    double  AnimDoubles[MAXPARAMS];
    char   *AnimStrings[MAXPARAMS];
    virtual int Controller() = 0;
  public:
    Animator();
    void RegisterAlgoEvt(const char *, const char *);
    int SendAlgoEvt(const char * ...);
    void RegisterView(BaseView *);
    void RemoveView(BaseView *);
    int Animate(int, int);
};
```

1.1 Entry points

The program driving the animation should call the RegisterAlgoEvt function and register each algorithm event prior to any of their transmissions. This registration handles their names and parameter types. Then, to actually send an event, use the SendAlgoEvt member function. This function stores the event name in the AlgoEvtName field, and the integer (double, string, resp.) trailing parameters in the corresponding member variables.

```
void
Animator::RegisterAlgoEvt (const char *name, const char *parampattern)
```

This routine registers the event with name *name* and a trailing parameter pattern. This is a string of d's (integers), f's (doubles), and s's (strings). For example, a *parampattern* of "ddfs" says that this event has two integers, a double, and a string, in that order. Currently, it is not allowable to overload a given name with multiple different parameter patterns.

```
int
Animator::SendAlgoEvt (const char * ...)
```

This routine actually transmits an algorithm event to an Animator. It first stuffs the name and trailing parameters into the variables as described above. In the parameter pattern "ddfs", AnimInts[0] gets the first int, AnimInts[1] gets the second int, AnimDoubles[0] gets the double, and AnimStrings[0] gets the string. It then calls the function Controller, which you should have defined in your derived Animator class. In there, you should check what event was just sent, and the call the appropriate View functions.

```
void
Animator::RegisterView (BaseView *)
```

This routine and the next two are used when multiple animation views will be in service for one Animator. This routines registers a View or StaticView (subclasses of BaseView) with its Animator. That is, this routine tells an Animator that a View exists, and must be called when animating via the following routine. Each View should call this routine, passing its "this" pointer as the parameter, at start-up. Make sure to read about and use the next function, RemoveView, if you use RegisterView.

```
void
Animator::RemoveView (BaseView *)
```

This routine should be used when you want to delete a BaseView (View or StaticView) and the View has earlier been registered with its Animator by using the RegisterView procedure. Make sure to call this routine BEFORE you delete the View. Essentially, we need to remove this View from the list of Views associated with the Animator. Consequently, subsequent calls to Animator::Animate will not utilize this View.

```
int
Animator::Animate (int start, int len)
```

This routine is used to animate multiple animation Views (View or StaticView) simultaneously. When this routine is called, each View that has been registered with RegisterView, will have its individual View::Animate routine called. Control will be interleaved among the Views on a frame-by-frame basis, thus providing the illusion of simultaneity. The parameter *start* provides the time from which to start animating (the first time passed to the individual Views), and the parameter *len* specifies how many frames should be generated. The routine returns the time after the animation has taken place ($start + len$). **Note:** you do not need to use this routine if your animation only has one view.

1.2 Example

Below is an example of a correctly defined subclass of Animator.

```
class MyAnimator : public Animator {
  private:
    View1 v1;   // see next section
    View2 v2;
  public:
    int Controller();
}
```

The controller might look like

```
int MyAnimator::Controller()
{
   if (!strcmp(AlgoEvtName,"Init")) {
      v1.Start(AnimInts[0],AnimDoubles[0]);
      v2.Init(AnimInts[0],AnimDoubles[0]);
   }
   else if . . .
}
```

2 Animation Views

An animation View is a particular graphical perspective of a program. Each resides in its own X window. A series of animation frames will be generated in the View. The frame number will be thought of as the animation time. This value is defined in the class member *time*, which is initialized to zero. Polka provides an abstract base class called BaseView. This is simply a placeholder—you should never create one of these. The two subclasses of BaseView that you *can* use are called a View and a StaticView. A View can have a variety of graphical objects inside of it. 99% of the time, this is what you should use. A StaticView is like a color bitmap. You simply can draw into it.

2.1 View

A View is what is used in Polka to generate animations the vast majority of the time. Actually, you don't use a View directly, you subclass it yourself. In addition to the functions already provided for a View, the View should also have individual animation scenes (such as Input, Compare, Exchange, and InPlace for a sorting view) placed in it.

Views contain a real-valued animation coordinate system. Typically, views begin with x and y ranging from 0.0 to 1.0, with the origin at the lower left corner and increasing from left-to-right and bottom-to-top. If the window isn't square initially, one of the dimensions will be skewed, and for instance, circles will look like ellipses. To get a non-square window to have a similar x-y aspect ratio, use the routine SetCoord.

The View base class is defined as follows:

```
typedef void (*TIMER_CALLBACK)(View *, int);

class View : public BaseView {
   protected:
      int     time;
   public:
      View();
      ~View();
      void SetDebug(int);
      void SetCoord(double, double, double, double);
      void SetBgColor(const char *);
      int Create(const char *title="Polka", RefreshMode rm=CoordStretch,
                 int width=0, int height=0);
      int Animate(int, int);
      int Simulate(int, int);
      void Restart();
      void Map();
      void UnMap();
      void Refresh();
      int CheckInput();
      int PickCoord(double&, double&);
      int PickAnimObject(class AnimObject* &);
      AnimObject *ObjectAt(double, double);
      void RegisterTimerCallback(int, TIMER_CALLBACK, void *);
      void SetRedrawAll(int);
      int GetDebug();
      char *GetBgColor();
      int GetMapped();
      void GetCoord(double&, double&, double&, double&);
      void GetDimension(int &, int&);
};
```

2.1.1 Entry Points

```
void
View::SetDebug(int d)
```

This routine sets the debugging output level (0-off, 1-on). The default is off. It's sometimes a good idea to have it set as you first develop an animation.

```
void
View::SetCoord(double lx, double by, double rx, double ty)
```

This routine is used to change the coordinates of a View that are visible. It can be called before or after the View::Create routine is called. It can be used to smoothly pan or zoom the window by making repeated, incremental calls to the routine. Note, however, that this call does not synchronize with the animation loop. The two are independent. To get panning or zooming that are synched with the animation refresh cycle, use the ALTER_LL and ALTER_UR Polka Actions.

```
void
View::SetBgColor(const char *)
```

This routine changes the background color of the View. Any valid X color name can be passed to the routine and the background color will immediately change. This routine should only be called after View::Create has been called, but it can be before any animation frames have been generated.

```
int
View::Create (const char *title="Polka", RefreshMode rm=CoordStretch,
              int width=0, int height=0)
```

This is the routine that puts up the X window housing the animation view. It MUST be the first routine called for the view, except for SetDebug or SetCoord which can precede the Create call. If a string is passed in as the first parameter, that string will be displayed in the window manager's title bar (if there is one) for the View. If none is passed in, the word "Polka" will appear there. The second argument should take on the value `CoordStretch` (the default) or `ConstantAspect`. This parameter specifies how the View contents will be displayed when the viewer manually resizes the window containing the View. If `CoordStretch` is chosen, then four corner window coordinates of the View always stay the same on a resize. This effectively stretches or shrinks the AnimObjects inside the View window. If `ConstantAspect` is chosen, the lower left coordinate stays the same, but the upper right coordinate is changed to keep the same exact pixels-to-coordinates aspect ratio that was in place before the resize. The final two arguments specify the width and height of the View in pixels at start-up. Note that the default values 0 are used just to make the value be taken from a Polka resource file. Even if no resource file exists, the default value of a $512X512$ View will be used. The routine returns 0 if it was not able to create the window successfully.

```
void
View::Restart()
```

This routine is used to "restart" a View from its start again. The background is changed to white again, the window coordinates are reset to the defaults, the *time* variable is set back to 0, and all the memory space used by the View, such as AnimObjects, is deleted and freed up. The window's size does not change, however, and the window borders and dressing stay visible and mapped.

```
void
View::UnMap()
```

This routine will remove a View from being shown (clears it off the screen altogether). You can still have the View do animation, create AnimObjects, etc., while it is unmapped. Then, when the window becomes mapped again, it will reflect the proper later state. You are allowed to call UnMap before you call Create. In this case, the View starts invisible and animations will proceed "behind the scenes." At the first Map call, the View will become visible and show the current animation state. Important Note: you should not call routines such as `PickCoord` that request input while a View is unmapped.

```
void
View::Map()
```

This routine brings a View back to being visible if it has been unmapped. Any changes done to the View in the meantime will be reflected when it returns.

```
void
View::Refresh()
```

This routine refreshes the View's window.

```
int
View::CheckInput()
```

This routine goes to the X event loop and processes any user input events pending for the window. It is useful at the very end of a program's execution to allow the Polka View to stay mapped and visible to the user. That is, place a call to `SendAlgoEvt` in a `while(1)` loop at the end of your program, where the response to the AlgoEvt is simply a call to this routine. (Note that this routine is automatically called by Polka as it generates animation frames. There is no need to call it during normal execution.) The routine always returns 0.

```
int
View::PickCoord(double& x, double& y)
```

This routine should be used in an animation scene to block and wait for the end-viewer to select, with the mouse, an x, y coordinate in the animation view. This coordinate can then be used, for example, to place an object as the endpoint of a target motion, etc. The routine returns 1 if the pick was made successfully.

```
int
View::PickAnimObject(AnimObject* &ao)
```

This routine should be used in an animation scene to block and wait for the end-viewer to select, with the mouse, an existing AnimObject. When the user selects a point, the highest-viewing-plane-level AnimObject that was picked will be returned through the parameter *ao*, and the routine returns 1. If no AnimObject was picked, the routine returns 0. Note that objects such as rectangles and circles respond exactly to their boundaries. Lines count as being picked if you click very close to the line. For other objects such as ellipses, text, splines and polylines, the object is considered to be picked if the mouse selection was within the object's bounding box.

```
AnimObject *
View::ObjectAt(double x, double y)
```

This routine returns the topmost AnimObject who lies at the given x, y position. Look at the routine `PickAnimObject` to learn about how AnimObjects respond to picks. If no AnimObject is at the specified position, the routine returns NULL. This routine is useful when you need to use PickCoord to get a position for some reason, but then you subsequently need to find out if the Pick was on an AnimObject.

```
void
View::RegisterTimerCallback(int framenum, TIMER_CALLBACK tcb, void *data)
```

This routine registers a user callback routine that will be called synchronously after the given number animation frame (available via the *time* View data member) has been generated. Any number of these callbacks can be registered and multiple ones can be registered for the same time. In that case, each will be called in an inverse order of their registration. Your callback routine should be defined in the following form:

```
void
tcb(View *v, int frametime, void *data);
```

Your routine is passed the View pointer, the current frame time, and the data field (which you passed in to RegisterTimerCallback) as parameters. This frame will be greater than or equal to the one you specified as the *framenum* param. (If somehow you skip frames with your Animate calls, your callback function is called on the first frame after that number occurs.)

Among other things, this routine is useful for C++ `delete`-ing AnimObjects after their AnimObject::Delete operation has been carried out at a particular time. IMPORTANT: You are not allowed to call View::Animate or View::Simulate from inside your timer callback routine. Any such attempt will be a no-op.

```
int
View::Animate(int start, int len)
```

This routine generates a set of animation frames, those starting with time=*start* and proceeding for *len* frames. In the particular animation scenes, you should have programmed AnimObjects to have certain desired behaviors during the frames being generated. The routine returns the time subsequent to the animation action (i.e. , start + len).

Note that it is undefined what will happen if you try to animate at a time in the past, that is, prior to a time to which you have already animated. For example, if you call Animate(3,1) then call Animate(1,1), weird things may happen on the second call. Note that if an AnimObject has been programmed to change at time i, and then you animate to time $i-1$ and then skip to time $i+1$, no extra animation frames will be generated (there won't be an ith frame) but the changes to the object that should have occurred at time i will be batched into the changes at the next subsequent frame, i.e., $i+1$ here.

```
int
View::Simulate(int start, int len)
```

This routine Works just like View::Animate, except that it does not generate the animation frames on the display. That is, it is like simulating the animation to occur: AnimObjects are updated according to their programming, but only their internal data structure attributes and values are modified, not the corresponding View. In essence, this routine provides a way to fast-forward past a number of animation frames. Typically, you should call View::Refresh after you have used Simulate and want to return back to calling Animate. The routine returns the time subsequent to the animation action (i.e. , start + len).

```
void
View::SetRedrawAll(int mode)
```

This routine can change Polka into a mode in which can potentially be faster, but it is not for all situations. Normally, when an AnimObject moves from one frame to another, we must erase it from its old position (the old frame) and then draw it in at the new position. This routine sets Polka into a mode (when the *mode* parameter is 0) where the previous position is not erased. Therefore, an AnimObject moving will seem to leave a ghost-like trail of itself. So, when is this useful? The "not redraw all" mode is useful when you have a View in which you only create new AnimObjects, but then they are static and never move. You simply only add new objects. To return to a normal (default) mode when everything is redrawn, call this routine with parameter 1.

```
int
View::GetDebug()
```

This routine returns the current debugging status.

```
int
View::GetMapped()
```

This routine returns whether the view is currently visible (returns 1) or invisible (returns 0).

```
void
View::GetCoord(double& lx, double& by, double& rx, double& ty)
```

This routine returns the current coordinates of the four sides of the View.

```
void
View::GetDimension(int& wid, int& hei)
```

This routine returns the current dimensions (in pixels) of the View.

```
char *
View::GetBgColor()
```

This routine returns the current background color of the View.

2.1.2 Example Definition

Below is a correctly defined subclass of a View class.

```
class View1 : public View {
   public:
      View1() { max=min=0;};
      int Init();
      int Input(int,int);
      int ExScene(int);
   private:
      int values[250],max,min;
      Rectangle *blocks[250];
      Loc *spots[100];
};
```

In the public section we have created the individual animation scenes making up the animation. In the private section, we have "global" variables that will be manipulated in the animation scenes.

An example View animation scene might look like

```
int
View1::ExScene(int i)
{
   Action a("MOVE", CLOCKWISE);
   int len = blocks[i]->Program(time, &a);
   time = Animate(time, len);
   return(1);
}
```

Note how the function manipulates the member variable *time*

2.2 StaticView

A StaticView differs from a View in that it does not maintain a display list of graphical objects. Rather it works like a drawing easel where you can keep on drawing new patterns on top of the old ones. So, in terms of functionality, it has much less than a View. But it's power comes from the fact that it does not maintain a display list of AnimObjects. Consequently, it is faster, and it does not use the memory space that a regular View does. You should use this when you just need an abstract drawing or sketching area.

Like a View, you don't use a StaticView directly. You subclass it. In addition to the functions already provided for a StaticView, your view should also have individual animation scenes (such as Input, Compare, Exchange, and InPlace for a sorting view) placed in it.

StaticViews, unlike Views, contain a integer-valued (pixel) animation coordinate system. StaticViews begin with x, y coordinates of (0,0) in the lower left corner and increasing from left-to-right and bottom-to-top.

The StaticView class is defined as follows:

```
class StaticView : public BaseView {
   protected:
      int     time;
   public:
      StaticView();
      ~StaticView();
      void SetDebug(int);
      int Create(const char *title="Polka", int width=0, int height=0);
      int Animate(int, int);
      int Simulate(int, int);
      void Restart();
      void Map();
      void UnMap();
      void Refresh();
      int CheckInput();
      int GetDebug();
      int GetMapped();
      void GetDimension(int &, int&);

      void DrawPoint(int, int, int, const char *);
      void DrawLine(int, int, int, int, int, const char *);
      void DrawRectangle(int, int, int, int, int, const char *);
      void DrawEllipse(int, int, int, int, int, const char *);
      void DrawText(int, int, int, const char *,const  char *, const char *);
};
```

2.2.1 Entry Points

```
void
StaticView::SetDebug(int d)
```

This routine sets the debugging output level (0-off, 1-on). The default is off. It's sometimes a good idea to have it set as you first develop an animation.

```
int
StaticView::Create (const char *title="Polka", int width=0, int height=0)
```

This is the routine that puts up the X window housing the animation view. It MUST be the first routine called for the view, except for SetDebug which can precede the Create call. If a string is passed in as the first parameter, that string will be displayed in the window manager's title bar (if there is one) for the View.

If none is passed in, the word "Polka" will appear there. The final two arguments specify the width and height of the View in pixels at start-up. Note that the default values 0 are used just to make the value be taken from a Polka resource file. Even if no resource file exists, the default value of a $512X512$ View will be used. The routine returns 0 if it was not able to create the window successfully.

```
void
StaticView::Restart()
```

This routine is used to "restart" a View from its start again. The background is changed to white again, the window coordinates are reset to the defaults, the *time* variable is set back to 0, and all the memory space used by the StaticView is deleted and freed up. The window's size does not change, however, and the window borders and dressing stay visible and mapped.

```
void
StaticView::UnMap()
```

This routine will remove a View from being shown (clears it off the screen altogether). You can still have the View do animation, while it is unmapped. Then, when the window becomes mapped again, it will reflect the proper later state. You are allowed to call UnMap before you call Create. In this case, the View starts invisible and animations will proceed "behind the scenes." At the first Map call, the View will become visible and show the current animation state.

```
void
StaticView::Map()
```

This routine brings a View back to being visible if it has been unmapped. Any changes done to the View in the meantime will be reflected when it returns.

```
void
StaticView::Refresh()
```

This routine refreshes the View's window.

```
int
StaticView::CheckInput()
```

This routine goes to the X event loop and processes any user input events pending for the window. It is useful at the very end of a program's execution to allow the Polka View to stay mapped and visible to the user. That is, place a call to `SendAlgoEvt` in a `while(1)` loop at the end of your program, where the response to the AlgoEvt is simply a call to this routine. (Note that this routine is automatically called by Polka as it generates animation frames. There is no need to call it during normal execution.) The routine always returns 0.

```
void
StaticView::DrawPoint(int time, int x, int y, const char *color)

void
StaticView::DrawLine(int time, int x, int y, int xsize, int ysize, const char *color)

void
StaticView::DrawRectangle(int time, int x, int y, int wid, int hei, const char *color)
```

```
void
StaticView::DrawEllipse(int time, int x, int y, int xrad, int yrad, const char *color)
```

```
void
StaticView::DrawText(int time, int x, int y, const char *string, const char *font, const char *color)
```

These are the drawing routines for the StaticView. The first parameter to each is the time (animation frame number) at which the drawing should occur. The subsequent parameters are the specifics of the drawing, and they should be pretty self-explanatory. Think of the StaticView as a painting easel or a colored bitmap. Each drawing operation simply sets some pixels in the view.

```
int
StaticView::Animate(int start, int len)
```

This routine generates a set of animation frames, those starting with time=*start* and proceeding for *len* frames. In the particular animation scenes, you can schedule different shapes to be drawn during the frames being generated. The routine returns the time subsequent to the animation action (i.e. , start + len).

Note that it is undefined what will happen if you try to animate at a time in the past, that is, prior to a time to which you have already animated. For example, if you call Animate(3,1) then call Animate(1,1), weird things may happen on the second call. Note that if a shape has been scheduled to be drawn at time i, and then you animate to time $i-1$ and then skip to time $i+1$, no extra animation frames will be generated (there won't be an ith frame) but that shape will appear in the StaticView in the next subsequent frame, i.e., $i+1$.

```
int
StaticView::Simulate(int start, int len)
```

This routine Works just like StaticView::Animate, except that it does not generate the animation frames on the display. That is, it is like simulating the animation to occur: Shapes are implicitly drawn, then they will appear in subsequent animation calls. In essence, this routine provides a way to fast-forward past a number of animation frames. Typically, you should call StaticView::Refresh after you have used Simulate and want to return back to calling Animate. The routine returns the time subsequent to the animation action (i.e. , start + len).

```
int
StaticView::GetDebug()
```

This routine returns the current debugging status.

```
int
StaticView::GetMapped()
```

This routine returns whether the view is currently visible (returns 1) or invisible (returns 0).

```
void
StaticView::GetDimension(int& wid, int& hei)
```

This routine returns the current dimensions (in pixels) of the StaticView.

2.2.2 Example Definition

Below is a correctly defined subclass of a StaticView class.

```
class View1 : public StaticView {
   public:
      View1() { max=min=0;};
      int Init();
      int Input(int,int);
      int ExScene(int);
   private:
      int values[250],max,min;
};
```

In the public section we have created the individual animation scenes making up the animation. In the private section, we have "global" variables that will be manipulated in the animation scenes.

An example StaticView animation scene might look like

```
int
View1::ExScene(int i)
{
   DrawLine(time, 20,20, 100,200, "red");
   time = Animate(time, 10);
   return(1);
}
```

Note how the function manipulates the member variable *time*

3 Objects in an Animation View

POLKA provides three basic types of objects which are created and manipulated in a view to create an animation. A Loc is a logical (x,y) position or location within an animation View. An AnimObject is a graphical object such as a line, circle or rectangle that changes to simulate animation. An Action is a typed change or modification, such as a movement along a path or a change to a different color. Below we describe each class in more detail.

3.1 Loc

A Loc(ation) is an (x,y) position within a View window. Locs are convenient for placing AnimObjects or having movements run relative to them. The x and y positions in a Loc are actually double values.

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.

Below is the class definition for a Loc.

```
class Loc {
   public:
      Loc();
      Loc(double x, double y);
      ~Loc();

      Loc *Copy();
      Loc *Modify(double x, double y);
      void Alter(double, double);
      double XCoord();
```

```
      double YCoord();
      int operator == (Loc &loc);
};
```

3.1.1 Entry Points

```
Loc::Loc()
```

This constructor creates a location with position (0.0, 0.0).

```
Loc::Loc(double x, double y)
```

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

```
Loc *
Loc::Copy()
```

This routine returns a copy of the location instance upon which it is invoked.

```
Loc *
Loc::Modify(double x, double y)
```

This routine creates a new Loc that has coordinates equal to the invoked Loc's coordinates plus the parameter values provided as arguments. (The values are simply added together).

```
void
Loc::Alter(double newx, double newy)
```

This routine alters the values stored within the location to be the new parameters passed in.

```
double
Loc::XCoord()
```

This routine returns the x coordinate of the invoked upon Loc.

```
double
Loc::YCoord()
```

This routine returns the y coordinate of the invoked upon Loc.

```
int
Loc::operator == (Loc &loc)
```

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

3.1.2 Groupings of Locs

We have also provided two convenience functions which allow you to create lines and grids of Locations respectively. The following class structures are required:

```
class Loc1Desc {
   public:
      int num;
      double fx,fy;    // from
      double tx,ty;    // to

      Loc1Desc();
      int Make(Loc*[]);
};

class Loc2Desc {
   public:
      int rows;
      int cols;
      double llx,lly;    // lower left
      double urx,ury;    // upper right

      Loc2Desc();
      int Make(Loc*[]);
 };
```

A Loc1Desc has members for the number of Locs to create, and the starting location (specified in *fx* and *fy*) and the ending location (specified by *tx* and *ty*) of the line of evenly spaced locations.

A Loc2Desc specifies the number of rows and columns to create plus the lower left and upper right values of the bounding box inside which to place the Locs.

In order to use these structures, you should create one and then fill in the individual fields to meet your particular needs.

Entry Points

```
int
Loc1Desc::Make(Loc *locs[]);
```

This routine is invoked for a line description, and creates a sequence of Locs according to its member values. The created Locs are returned in locs[0] to locs[num-1]. The routine returns 0 if it could not create the Locs (e.g., your Loc1Desc was bad).

```
int
Loc2Desc::Make(Loc *locs[]);
```

This routine is invoked upon a grid description, and creates the 2-d array of Locs (proportionally spaced in x and y) according to the specifications in the description. The Locs are loaded into locs starting in the lower left and proceeding row-by-row. For example, locs[0] contains to lower-left spot, locs[1] contains the Loc just to the right of the lower left corner, and locs[cols] contains the Loc just above the lower left corner spot.

3.2 AnimObjects

An AnimObject is a type of graphical object that's visible in an animation window. By manipulating these objects, changing their positions, sizes, colors, visibilities, etc. a programmer builds an animation. Actually, AnimObject is the super class of all the different object types, so an AnimObject itself is never created, only the individual subclasses. AnimObject provides a few important methods which are used for all the different derived types, plus it provides a number of virtual functions which must be written for each specific subclass. Below is the class definition AnimObject.

```
class AnimObject {
   public:
      AnimObject(const AnimObject&);
      void Originate(int);
      void Change(int, AnimObject *, int useoldpos=0);
      void Delete(int);
      int Program(int, Action*);
      int ProgramContinuous(int, Action*);
      void RemoveContinuous(int, Action*);
      void StoreData(void *);
      void *RetrieveData();
      int SetCallback(CALLBACKFCT, void *);
      Attachment AttachTo(AnimObject *, PART, PART, double=0.0, double=0.0);
      int Detach(Attachment);
      virtual Loc *Where(PART);
      void GetValues(View**, int*, double*, double*, ...);
};
```

All AnimObjects (except the special Set object) have four fields in common: an associated View, their visibility, and x and y locations. These are the four parameters to the AnimObject constructor. Each subclass constructor must call this superclass constructor. All AnimObjects have their own "copy" constructors and destructors too. Note that when an AnimObject is constructed, it does not appear in an animation View automatically. The user must call the Originate routine to have the AnimObject appear. The parameter to Originate will specify at which animation frame number the object will appear. To remove an AnimObject from a View permanently, call the Delete routine and specify at which frame it should be removed. It is also possible to change the appearance of an object, from say a Line to a Rectangle, with the Change function. All AnimObjects are programmed to perform an Action (see below) through the superclass function Program. Objects retain their programming across Change calls. The Where operation returns the current location of a part of the object.

AnimObjects are designed (actually "will eventually be designed") to be user-extensible. That is, an animation designer should be able to add her/his own new subclass. For convenience, we provide a large collection of subclasses types. Each constructor allocates a graphical image and returns a handle to it.

3.2.1 Individual Subclasses

Line

The constructor for a line is

```
Line::Line(View *view, int vis, double lx, double ly, double sx, double sy,
     COLOR col="black", double wid=0.0, double style=1.0, int arrow=0);
```

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. *style* 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. *col* gives the color of the image (COLOR is typedef'ed to char*). It can be any valid X color name such as "PeachPuff" or it can be an RGB valued string such as "#F7A305" (this corresponds to the mapping "#RRGGBB"). On a black and white monitor, colors may be implemented as a fill pattern. 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

The constructor for a Rectangle follows:

```
Rectangle::Rectangle(View *view, int vis, double lx, double ly,
          double sx, double sy, COLOR c="black", double fill=0.0) ;
```

Rectangles are located by their lower left corner, with only positive sizes ranging up and to the right allowed. The *fill* parameter is a value between 0.0 and 1.0. 0.0 corresponds to an unfilled outline (the background color shows through on the inside) and 1.0 corresponds to 100 per cent solid fill in the given object color. At in-between fill values, the object color is mixed with white to get intermediate shades, that is, no background color shows through. In reality, forty graduated fill patterns are actually implemented for these in between values. At 0.5, for example, the object color alternates with white on every other interior pixel.

Circle

The class definition for a Circle follows:

```
Circle::Circle(View *view, int vis, double lx, double ly, double rad,
           COLOR c="black", double fill=0.0);
```

For circles, the location pair denotes the center of the circle, with the *rad* parameter defining the radius of the circle.

Ellipse

The class definition for an Ellipse follows:

```
Ellipse::Ellipse(View *view, int vis, double lx, double ly,
          double sx, double sy, COLOR c="black", double fill=0.0);
```

For ellipses, the location pair identifies the ellipse's center. The *sx* and *sy* values denote the ellipse's radii in x and y respectively.

Polyline

The class definition for a Polyline follows:

```
Polyline::Polyline(View *view, int vis, double lx, double ly, int vert,
            double vtx[], double vty[], COLOR c="black",
            double wid=0.0, double style=1.0, int arrow=0);
```

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 *vtx* and *vty* 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 *style* 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).

Spline

The constructor for a spline is a as follows:

```
Spline::Spline(View *view, int vis, double lx, double ly, int vert,
         double vx[], double vy[], COLOR c="black",
         double wid=0.0, double style=1.0);
```

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 *style* define the width and style of the spline just as in the line image type.

Polygon

The constructor for a Polygon is as follows:

```
Polygon::Polygon(View *view, int vis, double lx, double ly, int sides,
          double vx[], double vy[], COLOR c="black", double fill=0.0);
```

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.

Pie

The constructor for a pie slice is as follows:

```
Pie::Pie(View *view, int vis, double lx, double ly, double rad,
        double begangle=0.0, double angledelta=1.0, char *col="black",
        double fill=1.0);
```

The Pie AnimObject supports pie slices.The location pair defines the center of the pie and *rad* defines its radius. *begangle* defines the angle at which to begin drawing the slice. Angles start (0.0) from due east (just like radians in math) and they sweep out in a counter-clockwise fashion. The values 0.0 to 1.0 sweep out a complete circle. Note that outlines (fill=0.0) do not presently work correctly.

Text

The constructor for text is as follows:

```
Text::Text(View *view, int vis, double lx, double ly, COLOR col,
        const char *fname, const char *textstring, int center);
```

The location pair identifies the lower left corner of where the text will be placed if the *center* 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 *textstring* parameter gives the text to be displayed. Note that only the actual text itself is drawn and colored in. That is, regions within the bounding box of the text, but not the actual characters, show through in the background color (transparent mode).

Bitmap

The constructor for a bitmap is as follows:

```
Bitmap::Bitmap(View *view, int vis, double lx, double ly,
         int width, int height, char data[], COLOR fg, COLOR bg);
```

The location pair identifies the lower left corner of where the bitmap will be placed. The width and height specify their relative values (in pixels) of the bitmap. We use the X protocol for bitmap specification. The following data array is an array of chars, where each bit is 1-fg or 0-bg. The array *data* is an array of bytes with the low order bit of the array's first byte containing the leftmost pixel of the first row of the bitmap. This is the exact format of the X11 utility *bitmap(1)*, so it is wise to use that tool to generate your data arrays. If you are on a color display, the bitmaps 1's will be drawn in the color *fg* and 0's in *bg* color. On black-and-whites these are black and white respectively. If for any reason POLKA was unable to create your bitmap (e.g., your width, height, and data were messed up), it will generate a clear white pixmap for subsequent use.

Set

The constructor for a set is as follows:

```
Set::Set(View *view, int num, AnimObject *objs[]);
```

A set is a special kind of AnimObject that is simply a reference to a list of other AnimObjects. A set is useful for grouping AnimObjects together (e.g. , a rectangle, its outline, and a centered text name) and then programming and performing Actions on the set object to save time and space, such as moving that whole collection of objects around together. *num* identifies the number of objects in the following array parameter. Note that a set can be used as a parameter to another set creation. One particular AnimObject will only be thought of as being in a set once, however. (It really does work like a mathematical set.) If a member of a set is deleted, that's fine. Subsequent actions upon the set just won't even act like that object ever existed. The member elements of a set respond to Action programming just as they would if they weren't involved in the set. Any Actions performed on the set are simply performed in order upon the set element objects.

3.2.2 Entry Points

```
AnimObject::AnimObject (const AnimObject&)
```

Each AnimObject subtype has a "copy" constructor that follows the prototype of the one given above that can be used for creating a copy of an AnimObject. The new object is give the exact same parameters as the old one, but of course, it exists now on a viewing plane closer to the end-user. The new object does not receive copies of existing programming from the original object.

```
void
AnimObject::GetValues (View**, int*, double*, double* ...)
```

Each AnimObject subtype has a call that corresponds to its constructor. The call is used to retrieve all the current values of the attributes of the object. The parameters are the same as in the constructor, except that they are pointers. (String parameters just take a string again, a new value will be allocated into the string passed in.) If you pass in the value NULL for a parameter, that argument is ignored. This allows you to query only the parameter values that are pertinent to you.

```
void
AnimObject::Originate (int time)
```

This routine should be called after an object has been constructed, in order to add it to a View. The parameter *time* specifies the frame time which the object should first appear. If the View's time has already progressed past that time, the object will appear at the first new frame generated. Don't forget to call this function! It is a common bug, and should be thought of immediately if your objects don't seem to be showing up.

```
void
AnimObject::Delete (int time)
```

This routine is used to remove an object from an animation View. The parameter *time* specifies the animation frame when the object should be removed. Deleting a Set object does NOT Delete its constituent objects. Note: For any AnimObject type, only after the specified frame time actually occurs (via the Animate call) is it safe to delete (C++) the AnimObject pointer.

```
void
AnimObject::Change (int time, AnimObject *obj, int useoldpos = 0)
```

This routine allows you to change the appearance or type of an AnimObject dynamically. For example, suppose you have programmed a Rectangle object to move across a window, but then you discover that halfway across, it should change to a Circle. To do this, simply construct a new Circle AnimObject (but don't Originate it!) and then call Change on the Rectangle, providing the frame at which it should change. Note that objects retain any programming and Set relationships they have across Change calls. The final parameter allows you to have the new object appear at the spot where the old object was ($useoldpos = 1$) or have the new object appear at the location given in its construction ($useoldpos = 0$). Important: when an AnimObject changes to a different appearance at a particular frame, any actions programmed for that object at that frame occur PRIOR to the object changing.

```
int
AnimObject::Program (int time, Action *action)
```

This routine "programs" or "schedules" the given Action to occur to the referenced AnimObject at the given time or animation frame. Subsequent calls to View::Animate which cover the specified time(s) actually make the programmed actions occur. Note that multiple Actions can be programmed onto an AnimObject at the same time or on overlapping times or on non-overlapping times. The effects of Actions are cumulative: all modifications scheduled for a particular frame occur "in parallel" and the resultant effect is seen when that frame number is generated. When an Action is programmed into an object, the contents of the Action are copied. Therefore, it is safe to immediately delete the Action after the call to Program.

If an Action is programmed to occur to an object at time i, then for some reason time i is skipped over in calls to Animate, all the actions that should have occurred earlier will be batched into the changes at the first frame actually generated subsequent to time i.

```
int
AnimObject::ProgramContinuous (int time, Action *action)
```

This routine "programs" an action to commence on an object at a given time, and then the Action will continuously keep being applied until it is removed with the routine RemoveContinuous. Subsequent calls to View::Animate actually make the programmed actions occur. The semantics of this routine work similar to Program, except that the Action keeps getting cycled over and over. This routine is good for having a clock that keeps ticking or some object that keeps spinning in a circle, and will not stop until you tell it.

Note that for continuous Actions, any skipped or missed frames are disregarded. Each continuous Action keeps an internal notion of what was the last offset of the Action to have been animated. Whenever the next valid Animate call comes around, the subsequent offset is performed.

```
void
AnimObject::RemoveContinuous (int time, Action *action)
```

This routine will remove an Action (which must have been set up using ProgramContinuous, and is designated by the parameter *action*) from the list of continuous Actions happening for an AnimObject. The Action will be removed at the beginning of the time cycle designated by the parameter *time*. That is, the Actions for that frame number will not occur to the AnimObject.

```
void
AnimObject::StoreData (void *data)

void *
AnimObject::RetrieveData ()
```

These routines allow the developer to store a client data value within each AnimObject. Using only the AnimObject itself, this value can later be retrieved.

```
typedef  void (*CALLBACKFCT)(AnimObject* ao, void *data, double x, double y);
```

```
int
AnimObject::SetCallback (CALLBACKFCT cb, void *data)
```

This routine allows you to register a callback routine (of the type described in the typedef above) for this AnimObject. Whenever the AnimObject is picked by the viewer using the mouse (see the routine View::PickAnimObject for semantics of what it means to pick an AnimObject), then the associated callback routine is called. The routine is passed the AnimObject itself, the client data stored on the orginal SetCallback call, and the coordinate location of the pick. One use for this routine is to make simple buttons by creating Rectangle AnimObjects with callbacks.

```
Loc *
AnimObject::Where (PART p)
```

This routine returns a Loc* that corresponds to the location of the given part of the AnimObject upon which the function is invoked. Valid PARTs include PART_C (center), and the compass directions PART_NW, PART_N, PART_NE, PART_E, PART_SE, PART_S, PART_SW, and PART_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.

```
Attachment
AnimObject::AttachTo (AnimObject *target, PART dept, PART indept,
                      double xoff=0.0, double yoff=0.0)
```

This routine can be used to "attach" one AnimObject to another. Whenever the latter AnimObject, *target* is subsequently modified, the dependent AnimObject will adjust its position to retain the attachment. Hence, attachments provide simple constraints. More specifically, the *dept* PART of the attached object is always positioned such that it aligns to the *indept* PART of the attached-to object. The last two optional parameters provide a means to have an offset in this connection. Currently, an AnimObject can only be attached in one way to a single object. Multiple different AnimObjects can be attached to the same object, however. Repeated calls to attach an AnimObject simply override the previous attachment calls. Polka also will detect and short-circuit any circularities in an attachment chain. Attachments occur and are enforced in the order of the list of AnimObjects (which also defines the stacking order of the AnimObjects). The attachments for the last AnimObject (very bottom of any stack) are carried out first when generating a new animation frame, and this continues onto the first AnimObject in the list (topmost plane). Its attachments are done last. The routine returns an Attachment ID which can subsequently be used in the Detach call. Set AnimObjects are not allowed to participate in attachments.

```
int
AnimObject::Detach (Attachment a)
```

This routine removes an attachment between two AnimObjects. The routine should be invoked upon the same AnimObject that the AttachTo call was invoked upon. The parameter is the Attachment ID returned from the AttachTo call. If the AnimObject has no current attachment, or the Attachment ID is not the valid one for the AnimObject currently, 0 is returned. If a successful detach occurs, 1 is returned.

3.2.3 Structured layouts of AnimObjects

For designer's convenience, we have also included routines that lays out a row or column of AnimObjects. This is a very common occurrence in algorithm animation. The following routines do not provide any *additional* functionality not elsewhere in the toolkit. These routines are merely a convenient abstraction, designed to save time and headaches. Basically, you must describe what your line of images will look like, and provide an array into which the routine can then place the objects as they are created.

The layout of the objects is described through an AnimObjectGroup structure which is implemented as follows below. You never use an AnimObjectGroup directly–it is an abstract base class.

```
class AnimObjectGroup {
   public:                      // default values
     int horiz;                 // 1
     Align align;               // AlignBottom
     int vis;                   // 1
     char color[32];            // "black"
     Spacing spacetype;         // SpacingSame
     double spacing;

     int useints;               // 0
     int *intvals;
     int intmax;
     int intmin;

     int usedoubles;            // 0
     double *doublevals;
     double doublemax;
     double doublemin;

     AnimObjectGroup();
};
```

The pertinent subclasses, which you do use directly, are shown below.

```
class LineGroup : public AnimObjectGroup {
   public:                      // default values
     double linewidth;          // 0.0
     double linestyle;          // 1.0
     int arrow;                 // 0
     LineGroup();
     int Make(View*, Line *[], int, double, double, double, double);
};


class RectangleGroup : public AnimObjectGroup {
   public:                      // default values
     double fill;               // 0.0
     RectangleGroup();
     int Make(View*, Rectangle *[], int, double, double, double, double);
};


class CircleGroup : public CircleGroup {
   public:                      // default values
     double fill;               // 0.0
     CircleGroup();
```

```
    int Make(View*, Circle *[], int, double, double, double, double);
};
```

The appropriate way to use these routines is to 1) allocate a static Group object which will call the constructor implicitly 2) Set the values of the public data members to be what you want them to be 3) Call the **Make** function of the appropriate subclass.

Here are some details about the individual data members. If *horiz* is 1, the routine creates a row (running left-to-right). If it is 0, a column is created (running bottom-to-top). *align* can be AlignBottom, AlignTop, or AlignMiddle for a row, and AlignLeft, AlignRight, or AlignMiddle for a column. If *vis* is 1, the objects are initially visible, and vice-versa. *color* holds their color. *fill*, *arrow*, *linewidth* and *linestyle* are self-explanatory (see early subclass descs.) *spacetype* can be SpacingNone (objects jut against each other), SpacingSame (objects are separated by an equal, consistent amount of space, and everything fits snugly), SpacingAbsolute (the *spacing* field provides an absolute space width). If *useints* is 1, then *intvals* should hold an array of integers which will be used to scale objects' heights (row) or widths (column) for a rectangle and line or scale the objects' radii for circles. In this case, *intmax* and *intmin* determine the largest and smallest values in that area. This routine will scale so that min gives height=0.0 and max give height=largest bounding box provided to the function below. The *doubles* fields work the same exact way; they are provided just in case the values you want to scale against are doubles and not ints,. If both *use* fields are 0, all objects have the same height (width) equal to the bounding box as given in the function parameters.

Well, this seems like a lot to specify, but it's really not. All the fields have default values (shown in the class definition). The only ones you need to specify are those that are different from the defaults.

The actual grouping calls are as follows (this is the Line one):

```
int
LineGroup::Make(View *v, Line* l[], int n, double lx, double by,
                double rx, double ty)
```

The routine creates a row of objects according to the fields of class upon which it is invoked. *v* is the View this is being created in. *n* is the number of objects; *l* is an array at least that big. The *lx, by, rx, ty* specify the bounding box inside which the sequence of objects will be created.

3.3 Action

An Action is just what its name implies: an action. OK, let's describe it a little more. An Action encapsulates a modification to an AnimObject such as changing its position, size, color, fill, and so on. Once an Action is created, it can be "programmed" or "scheduled" onto an AnimObject to commence at a particular frame number (see `AnimObject::Program` earlier in the documentation). Multiple different Actions can be programmed onto an AnimObject at the same time, or a particular Action can be used to program many different AnimObjects.

An Action is composed of 1) an Action type, which can be something like movement, resizing, change in color, etc., and 2) a sequence of *dx, dy* offsets that we call a path. Just think of a path as a sequence of steps in the *x*-*y* space.

In a movement Action, the control points or offsets in the associated path define where the AnimObject 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 object appears to move along this path. The *length* of an Action is the number of offsets its path contains.

Paths are not only used for movement, however. Every Action utilizes its path component to define its exact changes to an object.

Below is the class definition for an Action:

```
class Action {
   public:
      Action();
      Action(const char *);
      Action(const char *, int, double [], double []);
      Action(const char *, int);
```

```
      Action(const char *, MOTION);
      Action(const char *, const char *);
      Action(const char *, Loc *, Loc *, MOTION);
      Action(const char *, Loc *, Loc *, int);
      Action(const char *, Loc *, Loc *, double);
      ~Action();

      int       Length();
      double    Deltax();
      double    Deltay();
      Action *AddHead(int);
      Action *AddTail(int);
      Action *DeleteHead(int);
      Action *DeleteTail(int);
      Action *Copy();
      Action *ChangeType(const char *);
      Action *Reverse();
      Action *Smooth();
      Action *Rotate(int);
      Action *Scale(double, double);
      Action *Extend(double, double);
      Action *Interpolate(double);
      Action *Example(Loc *, Loc *);
      Action *Iterate(int);
      Action *Concatenate(Action *);
      Action *Compose(Action *);
};
```

Quite imposing, isn't it? Well, not really, once you play with it a little. Basically, there are just lots of different member functions that allow you to make paths in lots of different ways. When you want a path for movement, certain functions are especially useful. When you want one for a color change, others may be helpful. Just remember that down underneath is simply a type (which is just a character string) and a list of relative offsets.

In all the constructors below (except first), the initial argument t identifies the Action type. This is simply a character string. The types that we have predefined and designated how AnimObjects will react to them are "MOVE," "RESIZE," "VIS," "COLOR," "FILL," "ALTER," "RAISE," "LOWER," "ALTER_LL," "ALTER_UR" and others. Simply designate this string when you create an Action. Below we described how each of these affects the different AnimObjects.

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

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

COLOR - change the AnimObject to the color indicated by the path.

ALTER - change the string shown for a Text AnimObject.

FILL - change the "fill" component of the AnimObject. This works differently for different types of AnimObjects. For rectangles, circles, ellipses, polygons, and pies, this transition alters the AnimObject's fill style value by the x value of the offsets in the path. The x value is simply added in. If the AnimObject'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 AnimObject constructors for information on width and styles). Extreme values are reset to 0.0 and 1.0 as in rectangles and circles.

RESIZE - resize the AnimObject along the path. The various types of AnimObjects 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 upper 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. Pies work similarly. 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 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 actions modify polylines, polygons, and splines by altering the relative position of a particular vertex on the AnimObject (except the one denoting the AnimObject's position) by the relative offsets of the path. As opposed to RESIZE, the transitions GRAB1-GRAB7 alter only one particular vertex in the AnimObject's definition. Think of grabbing that vertex and swinging it around while all the other points stay anchored. With a Pie object, the x component modifies the beginning angle of the pie (0.0-¿1.0 corresponds to a complete circle) and the y component modifies the delta angle. GRABs have no effect on other AnimObjects.

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

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

ALTER_LL - this modifies the coordinate value of the lower left corner of the View by adding the x and y offset values of the Action to it. The AnimObject provided is totally ignored (but just don't use a Set because the coordinate will be modified for every member of the Set). This Action is useful to pan and zoom the View window in synchronization with the animation loop.

ALTER_UR - this modifies the coordinate value of the upper right corner of the View. Using both the ALTER_LL and ALTER_UR together on the same path at the same time allows you to pan the View around.

Below is a further explanation of the different Action types and the number of offsets in the path of an Action:

- Any type of action

  ```
  Action(char *type)                                    [0 offsets]
  ```

  NOTE: By itself, this action will do nothing because it has no offsets. You must use AddHead() or AddTail() to add offsets. In fact, if you specify 0 offsets in any other constructor, you get an error message stating that the action is useless.

- COLOR, ALTER

  ```
  Action(char *type, char *attribute)                   [1 offset]
  ```

  Sets the color attribute of any object (COLOR) or the string attribute of a text object (ALTER).

- RAISE, LOWER, VIS

  ```
  Action(char *type, int offsets)
  ```

  If offsets == 0, see the note at the top.

  If offsets >= 1 with the RAISE or LOWER actions, this action will raise the object to the front or lower the object to the back.

If offsets >= 1 with the VIS action, this will cause the visibility of the object to alternate between visible and invisible. If offsets is even, the object will end up with the same visibility status it started with. If odd, the object's visibility status will end up opposite that which it started with.

This constructor can also be used with the MOVE, FILL, RESIZE, GRAB, ALTER_LL, and ALTER_UR actions, but since all the offsets will be null, an action constructed this way won't do anything.

- MOVE, FILL, RESIZE, GRAB, ALTER_LL, ALTER_UR

```
Action(char *type, Loc *from, Loc *to, int offsets)
Action(char *type, Loc *from, Loc *to, double distance)       [>= 1 offset]
Action(char *type, MOTION motion)                              [20 offsets]
Action(char *type, Loc *from, Loc *to, MOTION motion)          [20 offsets]
Action(char *type, int offsets, double dx[], double dy[])
```

If offsets == 0, see the note at the top.

The first two constructors always create offsets in a straight line. Note that these constructors can also be used with the FILL action.

The next two constructors can produce offsets in a line or a clockwise or counterclockwise pattern, depending on whether motion is STRAIGHT, CLOCKWISE, or COUNTERCLOCKWISE.

The last constructor can have any offsets you provide. Note that it is easier to use one of the other constructors if you want action in a straight line or want the fill pattern to change linearly.

These constructors can also be used with the RAISE, LOWER, and VIS actions, but since the values of the offsets will be ignored, it's easier to use the simpler constructor designed for those actions.

3.3.1 Entry Points

`Action::Action()`

This constructor creates an Action with type = 0 and no offsets.

`Action::Action (const char *t)`

This constructor creates an Action with a type, but no offsets.

`Action::Action(const char *t, int len, double dx[], double dy[])`

This constructor creates an Action that has length *len,* designated by the given sequence of dx and dy offset pairs.

`Action::Action(const char *t, int i)`

This function creates an action with *null* offsets. When the function is given an integer greater than zero, it creates an Action with that number of *null* offsets, i.e., each dx and dy equal to 0.0.

`Action::Action(const char *t, MOTION m)`

This function creates an action with a path in one of three basic types (motions), designated by the constants STRAIGHT, CLOCKWISE, and COUNTERCLOCKWISE, then it creates a special set of offsets. STRAIGHT moves right; CLOCKWISE moves clockwise right in an upward arc; COUNTERCLOCKWISE moves counterclockwise left in an upward arc. Each of the 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. These Actions 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 modification routines.

`Action::Action (const char *t,const  char *arg)`

This routine should be used to either change the color of an AnimObject or to change the string in a text AnimObject. It should be called with a first parameter of either "COLOR" or "ALTER", respectively. In the color case, it returns a one offset Action that will change an object to the color given as the *arg* parameter (an X11 valid color string). This will only work if the string t is "COLOR." In the alter case, the text string for the AnimObject will be changed to whatever string is in *arg*.

`Action::Action (const char *t, Loc *from, Loc *to, int steps)`

This routine returns an Action that proceeds in a straight line from the given *from* Loc* to the given *to* Loc* with the condition that the path will have the given number (*steps*) of equally spaced offsets.

Note that in this routine and the two following, the provided from and to locations need not have absolute screen coordinate meaning. Because an Action 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 Action 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. Typically, you want t to be "MOVE" in these three routines.

`Action::Action const (char *t, Loc *from, Loc *to, MOTION m)`

This routine returns a path with movement characteristics of the given type m, but that also begins at the location with logical x and y coordinates of *from* and proceeds to the location with logical x and y coordinates of the *to*.

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

`Action::Action (const char *t, Loc *from, Loc *to, double distance)`

This routine returns an Action that proceeds in a straight line from the given *from* Loc* to the given *to* Loc* 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 from and to, the path is only given one offset. This routine is useful to create Actions that will make AnimObjects move at the same velocity.

```
int
Action::Length ()
```

This routine returns the numbers of offsets in the invoked Action.

```
double
Action::Deltax ()

double
Action::Deltay ()
```

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

```
Action *
Action::AddTail (int num)
```

```
Action *
Action::AddHead (int num)
```

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

```
Action *
Action::DeleteTail (int num)
```

```
Action *
Action::DeleteHead (int num)
```

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

```
Action *
Action::Copy ()
```

This routine returns a new Action that has an exact duplicate list of offsets and same type as the invoked Action.

```
Action *
Action::ChangeType (const char *new)
```

This routine returns a new Action that has an exact duplicate list of offsets as the invoked upon Action, but that has a new type, defined by the string *new*.

```
Action *
Action::Reverse ()
```

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

```
Action *
Action::Smooth ()
```

This routine returns an Action of the same type as the given one, but that has a "smoother" path component. Essentially, each new path offset value is determined by averaging in the neighboring offsets' values. Currently, we use twice the designated offset plus the previous and subsequent offsets, divided by four.

```
Action *
Action::Rotate (int deg)
```

This routine returns an Action that corresponds to the rotation of the invoked Action 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.

```
Action *
Action::Scale (double dx, double dy)
```

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

```
Action *
Action::Extend (double dx, double dy)
```

This routine returns an Action 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.

```
Action *
Action::Interpolate (double factor)
```

This routine returns an Action in which the number of offsets is modified by a factor given by the *factor* parameter. If an Action with length 10 is interpolated with a factor of 2.0, the returned Action will have 20 offsets. If the factor is 0.3, the Action 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.

```
Action *
Action::Example (Loc *fromloc, Loc *toloc)
```

This routine returns an Action which "looks like" the invoked example Action, but which would move from the logical *fromloc* to the logical *toloc*. The created Action will have the same number of offsets as the invoked upon Action. The general flow of movement in the example Action is followed as closely as possible by maintaining the same ratios between control points in both Actions. Clearly, however, if the two Actions move in opposite directions, they will not look much alike. Typically, this routine will be used when one wants an AnimObject to end up in some specific position, with the AnimObject following some rough, given trajectory (from another Action) in order to get there.

```
Action *
Action::Iterate (int times)
```

This routine returns an Action which is an iteration of a given Action, one occurrence after another. The *times* parameter provides how many times the invoked Action should be repeated in the Action to be created.

```
Action *
Action::Concatenate (Action *a)
```

This routine creates an Action which is the concatenation of the given Action after the Action upon which invocation occurs. The first offset of an Action is relative to the last offset of the previous Action. The new Action retains the type of the referenced Action.

```
Action *
Action::Compose (Action *a)
```

This routine returns an Action which is the composition of the Action parameter a and the invoked upon Action. By composition, we mean a cumulative combination on an offset by offset basis. If a Action is thought of as a vector, the composition of Actions produces a new vector that has the same length as the originals and is like the vector sum of the original Action vectors. Only Actions with an equal number of offsets can be composed. Otherwise, the routine returns NULL.

Essentially, Action 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 Actions.

4 General routines

The routines in this section are generic and control some overall behavior of Polka across all Views.

4.1 Entry Points

```
void
PolkaPause ()
```

This routine forces a programmatic pause of an animation. That is, calling this routine is the equivalent of the user pressing the “Pause” button on the control panel while an animation is running. The user must then click the “Run” button manually to restart the animation (or it can be restarted programmatically via PolkaResume).

```
void
PolkaResume ()
```

This routine forces a programmatic resume of an animation. That is, calling this routine is the equivalent of the user pressing the “Run” button on the control panel while an animation is paused.

```
void
PolkaSpeed (int speed)
```

This routine allows for programmatic setting of the speed control on a Polka animation. The *speed* parameter can range from 0 (slowest setting) to 100 (fastest setting). When this routine is called, the speed control scrollbar in the control panel automatically updates itself.

The routines below are useful when a Polka animation is not the first Xt window to be started in a program. In other words, the Polka animation can be a subordinate window. These routines also allow you to disable Polka controlling the run/pause, step, and quit operations of the Polka control panel. Polka will still maintain the interface, but it can be set up to call your application when these buttons are pressed.

```
void
PolkaInitialize (Display *, XtAppContext, Widget)
```

You should use this routine if you need to run Polka as a subordinate Xt process or shell, as opposed to the very first one that starts up, which is the normal mode. If you wish to do this, simply acquire the X Display, application context and top-level application shell, then pass them to Polka through this routine. For example, if the application using Polka initially puts up a window with some buttons that the user can interact with, then this routine should be called and passed the appropriate parameters.

```
void
PolkaDisableControls()
```

Call this routine if you wish to disable Polka controlling the start/stop, stepping, and quitting of the animation through the control panel buttons. Polka normally does the semantic actions appropriate for each of these user interface buttons. If you wish to remove that, simply call this routine once at start-up. Not that Polka still will maintain the interface look, for example, toggling the stop/start button between showing "Run" and "Pause."

```
void
PolkaSetCallbacks(APP_CALLBACK runpause, APP_CALLBACK step,
         APP_CALLBACK delay, APP_CALLBACK close, APP_CALLBACK quit)
```

This routine allows you to register one of your own callback routines as the source routine to be called whenever a particular Polka user interface command is selected by the user. The first routine is called whenever the Start/Run/Pause button is picked, the second is called when the Step button is picked, the third is called when the Speed control scrollbar is moved, the fourth is called when any animation window is closed via the Close button on it, and the final is called when the Quit button is picked. All of these routines should be of the form of a standard X Toolkit callback routine, i.e.,

```
void (*APP_CALLBACK)(Widget w, XtPointer client, XtPointer call);
```

In the case of the Close callback, the closed View is passed as the client data.

```
void
PolkaReset();
```

This routine will reset the Polka control panel to appear as it intially does when the animation starts up. That is, it returns the Start button and delay slider to their initial values. The routine does not affect any modifications or callback routines you have set up for the buttons. Together with the `View::Reset` routine, this routine is useful for "restarting" an application or animation without having to shut down the program and rerun it.

Acknowledgments

Many people have assisted in the development of Polka. The system itself is a descendant of the Tango and XTango systems, owing much of its techniques and methodologies to ideas originally in those two systems. Eileen Kraemer helped sort out and refine the system's capabilities. Steve Ostlind helped fix some problems in the Text object. Steve Chapel helped improve this documentation.