A Practical Animation Language for Software Development

John T. Stasko
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280
stasko@prism.gatech.edu

Abstract
In this paper we describe a practical language for creating real-time, two-dimensional, smooth, color animations. Animation can be a valuable component in a variety of domains such as user interface design, on-line help information, and computer aided instruction. Unfortunately, current methods for creating animations tend to be ad-hoc, varying widely from application to application. Another weakness of current methods is that relatively little work has been done on the formal specification and semantics of graphics, especially animation.

Our animation language produces aesthetically pleasing, smooth imagery and is easy to learn and use. The language is based on four abstract data types: locations, images, paths, and transitions; animation designers create and modify objects of these types in order to produce animation sequences. In addition, we provide a precise specification and semantics for all the data type operations. This rigorous definition helps simplify animation design by formalizing the actions resulting from the operations. We have implemented a prototype algorithm animation system that utilizes our design language as its basis.

1 Introduction
The increasing availability and sophistication of workstations has made programmers come to expect more advanced imagery in user interfaces. With improvements in display hardware, animation is now a viable part of those interfaces. Unfortunately, animation design methods are usually ad-hoc, lacking conceptual support, and peculiar to the particular application. Often, such animations only consist of a repeated shuffle through slightly altered sequences of precomputed bitmaps.

What is needed is a practical graphics language containing primitives specifically tailored for animation. The language should be as simple as possible in order to encourage its use, and it should be general purpose, capable of displaying a wide variety of imagery. Another requirement for the language is that it have a precise semantics with formal specification. Although the merits of specification are universally acknowledged, relatively little work has occurred concerning specification of picture images, especially animated images.

We have developed a language for producing color, two-dimensional animation sequences. Our language supports aesthetically pleasing, smooth image changes, and includes animation primitives specifically created to simplify animation design and implementation. We also provide formal models and precise semantics to insure the consistency of design sequences. Our approach is object based: by creating and manipulating objects of four simple data types, programmers produce their desired animation actions.

The specific application area that our work addresses is algorithm animation, the production of animated visualizations of the abstractions, data, and operations in executing computer programs. Algorithm animations occur in real-time, without delays from image rendering. As a program runs, it provides information to its accompanying animation view and thus controls the imagery and display. Algorithm animation is particularly difficult because animations cannot be totally predefined—run-time data dictates how the animation must appear.

The application of our animation language need not be restricted to algorithm animation, however. Animation is currently becoming more evident in a variety of areas such as computer-aided design, user interfaces, and on-line help information. The advantages of a formal design paradigm over ad-hoc methods are clear; our language provides a practical example that can be used in these and other areas.

2 The Path-Transition Paradigm
A computer animation consists of a collection of graphical objects to which a collection of modifications such as movement, changes in color or size, and so forth, occur over time. We have identified a set of four abstract data types that encapsulate the constituents of an animation. These animation data types serve as the basis for what we term our “animation framework.” They include the graphical images on the screen, the locations they occupy, the transitions the images make, and the paths that modify the image transitions. Our animation design method is called the path-transition paradigm, based on the two data types that drive the animation actions. As part of each data type, we also provide a set of constituent operations. Defining an animation involves creating and manipulating specific instances of the data types using these operations.

To examine in more detail the way that the image, location, path, and transition data types are utilized, we will consider, as an example, a specific activity that might occur in an animation. Suppose that we have a rectangle and a line in the animation viewing area as in Figure 1.
We would like to design an animation that consists of the rectangle moving in a straight path over to the top of the line. The rectangle and line pictures are instances of the image data type. We will assume that some form of image creation operation put them in their current positions. Because we would like the rectangle to move to the top of the line, we will inquire the current positions of the center of the rectangle's lower edge and the top endpoint of the line. These coordinate positions are examples of the location data type. We can utilize the two locations as endpoints in a path that consists of a sequence of interpolated points between the two path endpoints; this path is an example of the path data type. The motion of the rectangle along the path provides an instance of the final data type, the transition. This is a very simple example of an animation activity and the way that the path-transition paradigm applies to it.

Our images and animations are laid out in a real-valued, infinite animation coordinate system. By designing in terms of real values instead of integer values representing pixels, animations can be designed to function regardless of the animation window size. In practice, our framework is designed to assist the production of animated views in windows on a bit-mapped graphics workstation. We will refer to such windows as animation windows. Therefore, the particular animation system implementation of the framework is responsible for mapping the animation coordinate system onto an actual animation window. One common example mapping is to create a square viewing area in the window with coordinate (0.0, 0.0) at the upper left corner and the coordinate (1.0, 1.0) at the lower right corner of the viewing area.

In the remainder of this section we will discuss each of the four animation data types in more detail. We also will list some of the operations that we have identified as being useful for creating the styles of animations we desire. To conclude the section, we will discuss our mapping mechanism from a computer process into the animation framework.

2.1 Abstract Data Types

2.1.1 Locations

A location is a position of interest within the animation coordinate system and is identified by an \((x, y)\) coordinate pair. The ability to save and reference particular locations is an important tool for animation design because geometric positioning of the animation entities is one of the most valuable methods to convey information from a program. We define the following operations on the location data type. The location creation operation LocCreate takes \(x\) and \(y\) coordinate values and returns a location object. To determine if two particular location instances reference the same coordinate values, we provide a location equivalence inquiry operation, LocEqual. For calculations involving particular location coordinates, we provide the LocX and LocY operations to reference the individual \(x\) and \(y\) coordinates of a location. Coordinates within a location can be changed through the use of the LocModify operation. This operation provides a general modification scheme as the two real values are simply added to the \(x\) and \(y\) coordinates of the given location.

2.1.2 Images

Animation is the process of giving movement and action to objects. In computer animation these objects are static pictures that undergo changes in location, size, color, and so forth, throughout frames of the animation to simulate action. In our animation framework, two types of picture objects called images, exist: primary images and composite images.

Primary Images

Primary images are the most basic types of images manipulated in the framework. Some common examples are lines, circles, rectangles and text. Restricting primary images to a specific format or a given set of primary image types would greatly restrict the animation framework. Consequently, we define a primary image by a set of parameters and a set of methods. Formally, we define a primary image as a 6-tuple \((t, o, v, P, R, d)\) where \(t\) is the primary image type, \(o\) is the image's position, \(v\) is the image's visibility, \(P\) is a set of parameters local to the image type, \(R\) is a set of transition methods, and \(d\) is a drawing method. The use of the local parameters \(P\) is necessary because specific types of primary images differ. For example, a line may have size, thickness, and color local parameters, whereas a circle may have radius, fill, and color parameters.

Images are modified by framework objects called transitions acting upon them. As part of a primary image definition, each image type must have a method in which the particular transitions affect the image parameters. Movement and visibility transitions affect each primary image type in the same manner: the position and visibility fields are modified. Primary images also must handle all the other types of framework transitions in a well-defined manner. For example, a circle may have its radius parameter modified by a resize transition, whereas a line may simply ignore a fill transition. The final component of the primary image definition is a draw method. This component translates a primary image from its parametric definition to a static, graphical picture image.

In an object-oriented sense, specific types of primary images are subtypes of a general primary image supertype. The primary image supertype consists of position and visibility parameters along with movement and visibility transition method handlers. Each type of image added to the framework inherits these fields from the supertype and also defines its own set of local parameters and methods. This object-oriented view of images is convenient for system designers implementing the animation framework.

Composite Images

Composite images create a new type of image that is a
collection of primary images usually having some geometric relationship to one another. The ability to coalesce a set of primary images into one new image type and repeatedly create instances of the image greatly extends the graphical display of the framework. Composite images are defined as a list of primary images in a local coordinate system. Positions of the primary images are all relative to the position parameter provided as the location of the composite image. For example, suppose we wish to repeatedly create an instance of a door in an animation. The definition in Figure 2 produces the accompanying composite image. Each line of the definition identifies a new primary image. The first two real numbers on a line indicate the location of the image; the next numbers indicate widths and heights for the rectangles or a radius for the circle.

Both primary and composite images are created via the image creation operation, ImageCreate. We also provide a bounding box oriented image location query operation called ImageLoc. Given an image and a compass position such as NorthWest or East or the special Center position, the operation returns a location object that corresponds to that current position of the image. This operation is useful as illustrated by the following example. Suppose two rectangle images exist, and we want to create an animation in which the first rectangle moves over on top of the second rectangle, like boxes stacked up, with their left edges aligned. To do this, we would identify the SouthWest part of the first rectangle and the NorthWest part of the second rectangle. Then, to carry out the desired animation, we would create a movement path for the first rectangle to traverse with endpoints at these two locations.

Another issue involving images is their relative layering in the animation coordinate system. If two rectangles of differing colors occupy the same location, we must determine which one will be considered to be on a plane closer to the viewer, and hence be visible. To establish consistency in the framework, we set up a relative ordering of images in order to handle this problem. When a new image is added, it is always placed in the top-most relative plane (closest to the viewer). We also include transition actions in the framework that place images in the top-most and bottom-most planes. We will discuss these types of consistency issues in more detail in Section 3.

### 2.1.3 Paths

Images that undergo changes throughout time produce animation. In our framework paths are the only parameters to changes (called transitions) other than the affected image; paths serve as the primary definitional aspect of the actions occurring in a transition. Using paths as directional routes for images in movement transitions is a natural, well-accepted notion. An important concept within our framework, however, is the fact paths are the only modifiers used for all types of transitions, including those such as changes in color, visibility, and size. By establishing a consistent model for transitions, each takes an image and a path parameter, but nothing else, we have simplified the components necessary for designing animations. Animation designers need not worry about remembering the exact set of parameters that a particular transition type uses.

Formally, a path is defined to be a finite, ordered sequence of real-valued \((x, y)\) coordinate pairs, where each pair designates a relative offset from the previous position. The initial coordinate pair of a path is offset from the path’s starting position. The length of a path \(p\) is the number of coordinate pairs it includes. This concept of a path is a direct descendant of the \(p\)-curve structure of Baucker[2]. A pictorial view of a sample path of length 3 is given in Figure 3.

Paths designate two-dimensional routes in an abstract coordinate system. For actions such as movement, we can think of the coordinate system as our animation framework coordinate system. It is important to understand, however, that paths traverse purely relative coordinate values. Once a path is created, it is used in a transition to apply to an image that has an absolute position. Particular relative offsets of the path then define how the image will change in the animation. Notice that a path also contains a time component in some sense. Each subsequent offset determines the location and appearance of an image in the next frame or time unit of the animation. Consequently, paths actually define three-dimensional modifiers of images in animations: the \(x\) and \(y\) dimensions of the coordinate system and the time scale of the animation itself.

Animation designers maintain control of the relative spacing between offsets in a path. In a graphical environment that supports fast image updates, utilizing paths with very small changes in \(x\) and \(y\) from offset to offset produces smooth, visually pleasing animation effects. Removing intermediate offsets speeds up a motion; adding offsets slows it down. We believe this explicit mechanism of describing the path of an image’s movement is a much more natural way to design animations than requiring repeated “image draw” calls on an implicit path.

The basic operation for creating paths, PathCreate, receives an ordered list of locations; each location defines the \(x\) and \(y\) relative offsets. Frequently, it is necessary to know the length of a path. The PathLength operation returns the number of offsets contained in a path. The PathDx and PathDy operations return the net change in \(x\) and \(y\), respectively, that the path traverses from start to finish in the animation coordinate system. For the path in Figure 3, PathDx would return 0.3 and PathDy would...
Although the PathCreate operation can produce every possible path, it is an inconvenient format for creating paths that will actually be used in the animation framework. For additional utility, we provide operations that create some basic types of paths by introducing other, more pragmatic, path creation operations below.

Because most image movements follow simple routes such as straight lines or simple arcs, we introduce three basic path types into the animation framework: straight, clockwise, and counterclockwise. Visualizations of the three path types are presented in Figure 4. We define a standard for these basic types so that each has length 20 and moves 0.2 distance units in x from the starting location to the ending location of the path. These values are somewhat arbitrary—they produce aesthetically pleasing motions in simple animations we designed to run in square animation windows with a side measuring 1.0 units. The PathMakeType operation creates a path in one of these types. Although the paths created by this operation can be used directly in a transition, most often the paths will be edited in some way by a path modification operation before actually being used.

The method for specifying arcs in the Metafont system[9] using intermediate points as well as enter and exit angles presents another possibility that we are considering for path creation in the future. The method would have to be supplemented with a technique for specifying the number of offsets in the arc, however.

In the framework we provide operations that copy, rotate, scale, and extend paths. The PathInterpolate operation adds or removes offsets from a path while maintaining its overall change in x and y. This operation can be used to control the relative speed of an action. Changes occur more quickly along a path with fewer offsets. We also provide operations to add or delete offsets from the head or tail of paths. The PathColor operation returns a path, which when used in a color transition, will change an image to a specified color.

One key issue in using paths in an animation occurs when using paths as routes for image movement. Image movement often takes the form of moving an image from its current location to some new location that depends upon particular run-time data. These two locations, the starting and ending points, typically cannot be determined until that particular instant before the movement transition occurs. Consequently, compiled paths defined at animation design time such as those from the PathMakeType operation, may not move the image to the desired position. Therefore, the framework must include operations that receive two locations and create a path that begins at the one location and ends at the other.

PathDistance is such an operation. In addition to two locations, it takes a distance parameter in order to define a path containing relative offsets spread apart by the given distance. An example path is given in Figure 5. The left path is the example path, and the right path is produced by PathExample given the indicated source and destination locations.

We define three operators, concatenation, iteration, and composition, on paths. Concatenation and iteration append copies of paths together in the manner suggested by their names, and composition combines paths on an offset by offset basis, much like vector addition.

2.1.4 Transitions

Locations, images, and paths have set the foundation for the animation framework. All that is missing is the final component, action. The logical units of action or change in the animation framework, called transitions, utilize a path to affect an image by modifying its position or appearance. Moving a line in a wavy path, altering the fill style of a circle, or shrinking a rectangle as it changes color are all examples of transitions. Additionally, we can think of the simultaneous occurrence of all three of these actions as a new, albeit more complex, transition.

A simple transition is defined by the 3-tuple $(t, i, p)$, where $t$ is the transition type, $i$ is the image being altered, and $p$ is the path to be applied in the transition.
tered, and \( p \) is a path argument modifier. Some common examples of simple transition types are move, resize, fill, color, and alter visibility. Like that done for primary images, we do not restrict simple transitions to a specific set of transition types, however. The general model of a simple transition allows the framework implementation to be extended when a new type of transition is needed. Note that simple transitions correspond roughly to methods in a strict object-oriented sense. We can think of sending a method to a particular image object.

The path data type plays an important role in the definition of a transition. As discussed earlier, the coordinate pairs making up a path correspond to the individual frames of an animation. In essence, the path argument defines the duration of a transition. Nevertheless, different types of transitions utilize path arguments in different ways. Move transitions use paths as routes for images to follow. Each coordinate pair of the path specifies the distance and direction that the image should move for the next animation frame. Visibility transitions disregard the specific \( x \) and \( y \) values of the coordinate pairs of a path; each coordinate pair is simply interpreted by toggling the image’s visibility in the next animation frame. Full transitions utilize the \( z \) component of a coordinate pair by adding its value to an image’s fill value, which ranges between 0.0 (outline) and 1.0 (completely filled).

If we decompose a path into its individual relative offsets and consider each along with the transition type and primary image components, we encounter the substructure that makes up a simple transition. This structure primitive, called a transition unit, is the atomic level action that occurs in an animation. Each transition unit defines the action to occur in order to generate the next frame of the animation. More formally, a transition unit is defined as a quadruple \((t, i, x, y)\), where \( t \) is the transition type, \( i \) is the image affected, and \( x \) and \( y \) are the particular offset arguments. Simple transitions are nothing more than ordered lists of transition units.

Three important transition types, delay, raise, and lower are always included in the framework. The delay transition type creates transitions with “empty” transition units in which no change occurs. The length of the path argument denotes the number of empty transition units to create. This transition provides an animation designer with a method for delaying and coordinating the relative times of animation actions. The transition type raise forces an image to the top-most layer of the animation coordinate system. Each transition unit in a raise transition will bring the image argument of the transition unit to the viewing plane closest to that of the user. The image does not change its location in the animation coordinate system. The lower transition works in the expected, opposite manner. Semantics of these operations will be discussed in Section 3.

The TransCreate operation creates simple transitions for subsequent use. Three transition operators then combine transitions to form new, more complex transitions. The Concatenation operator binds two transitions together into a single, new transition which corresponds to the latter transition argument commencing immediately after the initial transition argument is complete. That is, the two lists of transition units are appended to form one longer list. The iteration operator performs repeated concatenation of a transition with itself. The third transition operator, composition, is more complex.

Composition introduces a form of concurrent execution of transitions. When two transitions are composed, the animation actions they denote occur simultaneously. Consider two separate move transitions defined upon a circle and a line respectively, that each utilize the same path argument. Composing these transitions creates a new transition that exhibits each of the images moving by the same relative offset, as defined by the path, between each frame of the animation. In essence, between each animation frame two transition units will occur—one for the circle image and one for the line image. Let us define a transition moment as a list of transition units that occur “simultaneously” in one time unit, in order to generate a new frame of an animation. This allows us to formally define a general transition in the framework as an ordered list of moments. Notice that a simple transition contains moments with only one transition unit. Figure 6 provides examples of these different types of transitions.

The concurrency exhibited by TransCompose is accomplished by combining corresponding moments in the two transition arguments into a new, larger moment in the resulting transition. The two lists of transition units making up the first moment of each transition being composed are appended to form a longer transition unit list that is the first moment of the resulting composition transition. The process is repeated for all moments in the transitions. If two transitions containing an unequal number of moments are composed, the shorter transition is implicitly padded at the end with empty moments to make the composition one-to-one. A good way to intuitively think about these operations is that concatenation combines lists of transition moments end-to-end, whereas composition combines them side-to-side.

Composition provides a natural way to perform an animation where more than one object changes. For example, consider the exchange of the positions of two rectangles. In order to design this animation, first we create two move transitions, each corresponding to one of the rectangles moving to the other’s position. Then we compose the two transitions into a single, new transition that denotes their concurrent movements.

As a more sophisticated example, suppose we wish to define a transition that moves an image \( i \) along a path \( p_1 \) that has length 20. We also would like the image’s visibility to toggle during the middle 10 movements in the path. We shall use the following notation for the transition operators:

\[
t_1 \cdot t_2 \equiv \text{concatenate transitions } t_1 \text{ and } t_2 \text{ (\( i \) first),}
\]

\[
t^{\text{iterate}} \equiv \text{iterate the transition } t \text{ \( n \) times,}
\]

\[
t_1 \odot t_2 \equiv \text{compose transitions } t_1 \text{ and } t_2.
\]

If we create the paths

\[
p_1 = \text{PathNull(5)},
\]

\[
p_2 = \text{PathNull(1)},
\]

and the transitions

\[
t_1 = \text{TransCreate}(\text{delay}, i, p_1),
\]

\[
t_2 = \text{TransCreate}(\text{visible}, i, p_2),
\]

\[
t_3 = \text{TransCreate}(\text{move}, i, p_1),
\]

Composition introduces a form of concurrent execution of transitions. When two transitions are composed, the animation actions they denote occur simultaneously. Consider two separate move transitions defined upon a circle and a line respectively, that each utilize the same path argument. Composing these transitions creates a new transition that exhibits each of the images moving by the same relative offset, as defined by the path, between each frame of the animation. In essence, between each animation frame two transition units will occur—one for the circle image and one for the line image. Let us define a transition moment as a list of transition units that occur "simultaneously" in one time unit, in order to generate a new frame of an animation. This allows us to formally define a general transition in the framework as an ordered list of moments. Notice that a simple transition contains moments with only one transition unit. Figure 6 provides examples of these different types of transitions.

The concurrency exhibited by TransCompose is accomplished by combining corresponding moments in the two transition arguments into a new, larger moment in the resulting transition. The two lists of transition units making up the first moment of each transition being composed are appended to form a longer transition unit list that is the first moment of the resulting composition transition. The process is repeated for all moments in the transitions. If two transitions containing an unequal number of moments are composed, the shorter transition is implicitly padded at the end with empty moments to make the composition one-to-one. A good way to intuitively think about these operations is that concatenation combines lists of transition moments end-to-end, whereas composition combines them side-to-side.

Composition provides a natural way to perform an animation where more than one object changes. For example, consider the exchange of the positions of two rectangles. In order to design this animation, first we create two move transitions, each corresponding to one of the rectangles moving to the other's position. Then we compose the two transitions into a single, new transition that denotes their concurrent movements.

As a more sophisticated example, suppose we wish to define a transition that moves an image \( i \) along a path \( p_1 \) that has length 20. We also would like the image’s visibility to toggle during the middle 10 movements in the path. We shall use the following notation for the transition operators:

\[
t_1 \cdot t_2 \equiv \text{concatenate transitions } t_1 \text{ and } t_2 \text{ (\( i \) first),}
\]

\[
t^{\text{iterate}} \equiv \text{iterate the transition } t \text{ \( n \) times,}
\]

\[
t_1 \odot t_2 \equiv \text{compose transitions } t_1 \text{ and } t_2.
\]

If we create the paths

\[
p_1 = \text{PathNull(5)},
\]

\[
p_2 = \text{PathNull(1)},
\]

and the transitions

\[
t_1 = \text{TransCreate}(\text{delay}, i, p_1),
\]

\[
t_2 = \text{TransCreate}(\text{visible}, i, p_2),
\]

\[
t_3 = \text{TransCreate}(\text{move}, i, p_1),
\]
the desired animation is simply the following expression:

\[(t_1 \cdot t_2) \cdot t_3 \cdot t_4 \cdot t_5 \]

Once a transition has been defined, the TransPerform operation actually executes it. Intuitively, this command works by sequentially examining the moments making up a transition; the changes denoted by the particular transition unit lists in a moment all occur “simultaneously” between the frames of the animation. By processing moments, one transition unit list after another, the frames of the animation are generated.

### 2.2 Mapping Component

Usually, an animation is a representation of a process, a mapping from the programming world into a visual world. For example, a direct manipulation user interface represents objects and data in an application. In algorithm animation the animated view is a simulation of the abstract processes of a computer program.

In our animation framework we provide the connection facility from program objects to animation objects through associations. Associations allow designers to connect data objects such as a framework image, location, or data value of importance to a set of parameters received from a driving program or process. An association between a framework object and another framework object can also be created. Associations are identified by unique names, and they can utilize zero or more parameters.

More formally, an association is a method to store a data object such as an animation data type object or a data value. A typical implementation uses hashing. The hash key for the data object is the name of the association and a list of parameters. The number of parameters to each association is fixed. Typically, the parameters are the data values being mapped from a program. For example, to associate an integer array `a` of length 10 with ten rectangle images, we could create an association with name “IMAGE”, first parameter the initial memory address of `a`, and second parameter the particular array index position.

Animation designers are free to create new association names as needed. We also provide a small set of predefined associations: The “ID” association with two parameters is usually used as a general purpose data mapping facility. The “IMAGEAT” association takes one parameter, typically a location object, and stores the image object currently associated with that location.

Three functions perform all of the association duties within the framework. The Create function defines an association and the number of parameters it utilizes. An association must be defined with this function before it can be utilized. The Store function associates a data object with a key—the association’s name and a list of parameters. Attempts to store a data object in an association with the incorrect number of parameters will be considered an error; no data will be stored. The Retrieve function returns the data object associated with a given key. If no data object has been stored with the given key, a null value is returned.

### 2.3 Animation Scenes

For manageability reasons, in the framework we decompose an animation into a set of smaller logical components called animation scenes. The creation and breakdown of an animation into scenes is purely determined by the animation’s designer; there is no “correct” decomposition. Quite often, however, a logical separation of important actions presents itself, not unlike the breakdown of a large computer program into subroutines.

In the framework we model an animation scene informally as a parameterized procedure. The procedure body consists primarily of the animation data type operations and association mapping calls described in this section. Animation scenes also often require mathematical and control flow facilities. Parameters to scenes are received by value; they consist of integer, real and string values.

As a first implementation, rather than creating a new stand-alone language, we utilized a host language for our framework. We implemented the four abstract data types in the framework as user-defined types in the C programming language. We also implemented the operations upon the types as a package of calls available to animation designers. This allows us to utilize C procedures as our animation scenes; the mathematical and control flow facilities provided by C are sufficient for designing sophisticated animation actions.

Figure 7 provides an example of an animation scene as it appears under our implementation. The scene’s purpose is to retrieve two rectangles that were created and stored in a prior scene, and then exchange their positions in a smooth, continuous looping motion. We assume that the rectangles have been stored under an association “RECT” taking an ID and an index as parameters. This scene is often utilized in algorithm animations when illustrating the exchange of values in a sorting program.

### 3 Semantics of Animation Operations

As discussed earlier, the animation component of our framework includes the visual images presented in the display and the transformations they undergo as time passes. We model the animation component through an animation state defined by the 4-tuple \((L, I, P, T)\), where \(L\) is a set of locations, \(I\) is a set of images, \(P\) is a set of paths,
The sole semantic purpose of each of the animation framework operations except for ImageCreate and TransPerform is to modify the animation state. Most operations add a new location, image, path, or transition object to the corresponding field in the animation state. Some simply return a parameter value about a framework object and make no change to the animation state. The ImageCreate function, in addition to modifying the animation state, adds images to the front of the image configuration list. The TransPerform operation is even more complex, however, as it can modify the animation state, reorder the image configuration list, and add one or more of these new configurations to the animation list. The number of new configurations generated corresponds directly to the length of the path(s) that serves as an argument to the transition being performed.

Two related, primary semantic issues that arise in the framework are the ordering of transition units within a moment of a transition, and the ordering of images within the configuration. These issues are important because we want to provide consistency throughout the framework and its operations.

Because of the TransCompose operation, many transition unit actions can occur "simultaneously" in one moment of a transition between the frames of an animation. The appearance of simultaneity is given by displaying the images in the current configuration, modifying the images' parameters as dictated by the transition units in that moment, then redisplaying the images in the resulting configuration. The difficulty occurs in that at some level we must still apply the transition units in a serial manner within the moment. Two move transition units defined upon one image offer no problem. Either application ordering of the two results in the image ending up at the same location. But if two color transition units defined upon the same image, one red and one blue, exist in a moment, their application ordering is critical. Whichever one is applied last will determine the image's color in the subsequent configuration.

Fortunately, because the only method for combining transition units into a moment is the TransCompose operation, we can deal with this issue by defining the way TransCompose generates transition unit lists. This is done by simply declaring that when composing two transitions, those transition units in the first transition parameter precede those of the second parameter when forming transition unit lists. That is, given the operation TransCompose(tr1, tr2) where tr1 changes an image to red and tr2 changes an image to blue, following the performance of this composition transition, the image will be blue.

The second important semantic issue concerns the ordering of images in a configuration and subsequently in the viewing window. As noted previously, images at the beginning of the configuration list will appear to be closer to the viewer. But how do images establish positions within the configuration list? We solve this problem through two simple rules.

1. Whenever an image is created, it is added to the front of the configuration list.

2. Whenever a transition unit of type raise is applied to an image, the image is brought to the front of the configuration list.
Hence, barring any use of the raise transition, the initial creation order of images defines their relative ordering in viewing planes of the animation coordinate system. Below we provide a look at the semantic formalisms we use to specify our design language. Due to space limitations, we include only a brief example. For the complete definitions, see [13].

The four animation data types are specified as follows:

- **Location** \( lo : \text{x:Real, y:Real} \)
- **Image** \( im : \text{type:ImageType, pos:Location, \_\_\_:\text{Boolean, \{local parameters\}} \)
- **Path** \( pa : \text{offsets:List(Location), \_\_\_:\text{Boolean} } \)
- **Transition** \( tr : \text{moment:List(List(TransUnit)))} \)

The symbols \( [\) denote a list of some object, and the symbol \( \emptyset \) denotes an empty list.

Our definitions provide a means for a better understanding of the animation generation process. As stated earlier, previous animation work is often void of such analysis. We desired to describe the framework operations in detail, yet remain true to the framework goal of a straightforward design process. We chose techniques similar to those of Mallgren[10] to accomplish this.

Following Mallgren's axiomatic designations for describing operations upon graphic data types, we divide operations upon the animation data types into three categories. Generators produce objects of the type of the operation (location objects for location operations, etc.). Inquiry operations return objects of some type other than that of the operation. Typically, they examine an object of the type of interest and return some parameter about it. Finally, basic generators are subsets of the generators, and are a sufficient set of operations within a type to generate any object of the type.

Operations are denoted and defined in the following manner: Basic generator operations are identified by the "o" symbol preceding the operation name. Generator operations are defined by expressions consisting only of basic generators and simple algebraic manipulations, along with basic constructs such as if-then-else and while-do. For manipulations of objects such as paths and transitions consisting primarily of lists, we also utilize the list operations car, cdr, cons, append, and reverse. Inquiry operations are defined by providing their result upon objects generated by all of the basic generators of that data type.

As an example, consider the path data type. Its basic generator operation is PathCreate. All generator operations such as PathNull, PathColor, and PathExample, therefore, must be defined in terms of PathCreate.

Both generators and basic generators affect the animation state by adding a new object to a field in the animation state as discussed earlier. These manipulations can be defined as in the location example:

\[
\text{LocCreate}(x, y) \rightarrow \text{lo}[x, y] : (lo \in L) \cup S,
\]

where \( L \) is the location component of the animation state \( S \). The equation merely defines that the \text{LocCreate} operation adds a new location object with the given \( x \) and \( y \) values to the location component of the animation state. All generator operations within the framework function similarly, so we will omit these manipulations from the operation definitions. Note that inquiry operations do not modify the animation state at all.

We believe that the descriptive methods we have chosen are rigorous enough to fully detail framework operations, and are informal enough so as not to obscure what is really going on. The conditional

\[
\text{if } (x)
\]

where \( x \) is a list of a given type of object, returns true except when \( x \) is the null list. We will focus on the transition data type here to illustrate these methods.

**Data type Transition**

**Operations**

- \text{TransCreate}: \text{transType} \times \text{image} \times \text{path} \rightarrow \text{transition;}
- \text{TransConcatenate}: \text{transition} \times \text{transition} \rightarrow \text{transition;}
- \text{TransIterate}: \text{transition} \times \text{integer} \rightarrow \text{transition;}
- \text{TransCompose}: \text{transition} \times \text{transition} \rightarrow \text{transition;}
- \text{TransPerform}: \text{transition} \rightarrow (\{ \text{image} \}).

**Preliminaries**

Below we define how a transition is generated by the TransCreate operation. The operation produces a list of moments, which in this case, are each only one transition unit. Assume \( \text{im} = \text{ImageCreate(imageType, bool, lo, \{params\})} \), and \( \text{pa} = \text{PathCreate(lo)} \).

\[
\text{TransCreate} (\text{trType, im, pa})
\rightarrow \text{if } (\text{pa}) \text{ then }
\begin{align*}
\text{cons}( & (\text{trType, im, LocX(car(pa))),} \\
& \text{LocY(car(pa)), \emptyset } ),
\end{align*}
\text{if } (\text{tr}) \text{ then }
\begin{align*}
& \text{TransCreate}(\text{trType, im, cdr(pa)}) \\
& \text{else } \emptyset.
\end{align*}
\]

**Example Definition**

Once a transition has been created via any of the first four transition operations in order to produce a desired animation, it is carried out with the special TransPerform operation. As discussed earlier, TransPerform may modify the animation state and rearrange the configuration. Additionally, it is the only operation that adds configurations to the animation list. This functionality of TransPerform is detailed below. The symbol \( C \) refers to the current configuration.

\[
\text{TransPerform(tr)}
\rightarrow \text{if } (tr) \text{ then }
\begin{align*}
& \text{moment} = \text{car(tr)} \\
& \text{while } (\text{moment}) \text{ do }
\begin{align*}
& \text{trans_unit} = \text{car(moment)} \\
& \text{apply(trans_unit, C)} \\
& \text{moment} = \text{cdr(moment)}
\end{align*}
\end{align*}
\text{return( cons( C, TransPerform(cdr(tr)) ) )}
\else \emptyset.
\end{align*}
\]
The apply operation receives a transition unit which we denote as \((\text{transType}, \text{im}, x', y')\). It performs two main functions.

First, it affects the animation state by modifying the image's parameters (the image is a member of the \(I\) field of the animation state) as dictated by the transition type and the image type. For instance, given that \(\text{im}\) is a circle,

\[
\text{apply}( (\text{move}, \text{im}, x', y'), C) \Rightarrow \text{im}:[\text{circle}, (\text{loc}+x', \text{loc}+y'), \text{vis}, \text{radius}, \text{color, fill}] \in I.
\]

\[
\text{apply}( (\text{resize}, \text{im}, x', y'), C) \Rightarrow \text{im}:[\text{circle}, \text{loc}, \text{vis}, \text{radius}+x', \text{color, fill}] \in I.
\]

Second, when given a transition unit of type raise, the apply operation removes the image from its current location in the configuration and brings it to the front of the list. The lower transition works similarly.

\[
\text{apply}( (\text{raise}, \text{im}, x', y'), C) \Rightarrow C - \text{im}; \cons(\text{im}, C).
\]

All operations and manipulations within the framework occur in order to produce animation. We introduced the concept of a configuration, an ordered list of the active images, in order to characterize the relative ordering of image objects. It allows us to formally define an animation as a list of configurations.

The animation process begins with an empty list of configurations. As animation scenes are activated, individual framework operations modify the animation state and the current configuration as described in this section. The special \(\text{TransPerform}\) operation adds configurations as elements in the animation list as follows:

\[
\text{AnimList} \triangleq \text{append}(\text{AnimList, TransPerform}(\mathcal{E})).
\]

\(\text{AnimList}\) begins as a null list, and it grows each time \(\text{TransPerform}\) is called. After the final animation scene is processed, we are left with the list of configurations characterizing the particular animation that has occurred.

4 Prototype Implementation

The initial testing ground for our animation framework is an implementation of an algorithm animation system. Algorithm animation is the process of abstracting the data, operations, and semantics of computer programs, and then creating animated graphical views of those abstractions. Algorithm animation encompasses what has been known traditionally as program animation and data structure rendering. These two areas typically involve one-to-one mappings between program data and the images in the program's animation. Algorithm animation, however, is a broader term and involves program views that represent abstractions we would visualize when considering the semantics of the program. Although a strict definition of algorithm animation would not include what we would consider to be animated simulations, in practice we frequently can produce animated simulations using algorithm animation techniques. Some of the better known algorithm animation systems are Balsa[3], Animus[4], and Aladdin[8].

The system we have developed is called TANGO (Transition-based ANimation Generation). TANGO supports two-dimensional color animations in a window-based workstation environment. It includes line, rectangle, circle, ellipse, polyline, polygon, spline, and text image types, as well as move, resize, color, fill, visibility, raise, lower, and delay transitions. To drive TANGO animations, programmers supplement their programs with events that we call algorithm operations. As program execution occurs, events are sent out as interprocess messages to a central message server that passes the messages on to TANGO animation windows. Individual messages generate state transitions within TANGO and activate the appropriate animation routines. These animation routines are user-written animation scenes that use the data types and operations of our animation design language.

Figure 8 shows an example of a TANGO animation of a bubblesort. We have superimposed a series of frames to illustrate the animation.

TANGO currently runs on a network of workstations and uses tools from the Brown Workstation Environment[11], an application interface toolkit built on top of the X11 window system[12]. We have used TANGO to animate programs from a wide variety of domains such as sorting, searching, hashing, and graph and tree manipulations, and we have designed animated simulations of a producer-consumer ring buffer, the Towers of Hanoi problem, and the post office queuing problem. TANGO also is being incorporated into computer science courses as an aid for understanding algorithms. The system is available for distribution.

5 Related Work

Our work touches on a variety of related areas such as graphical specification techniques and animation languages. With regard to graphical specification, previous work has focused primarily on static imagery. As mentioned earlier, Mallgren provided one of the earliest efforts toward formal specification of graphic objects by introducing special graphic data types such as points, regions, geometric functions, and graphic transformations[10]. Using algebraic axioms, Mallgren formally defined all operations on the data types. Fiume utilized a somewhat
different approach to formal specifications for a non-trivial graphical object by attempting to specify formally a bitmap object and its relationship to an image object[6]. In his approach, bitmaps and images were modeled as abstract data types with lists of mathematically and set-defined operations.

The Dial animation language by Feiner, Salesin, and Banchoff utilized a simple, two-dimensional notation to let animators express parallelism[3]. Animators expressed the coordination of object's motions using a chart-like syntax in which the horizontal dimension representing time and the vertical dimension specifying individual animation actions. Taking an object-oriented approach to animation, Fiume, Tsichritzis, and Dami presented a language for expressing the temporal coordination of animated objects[7]. In their system, designers created animated objects that had special temporal behaviors with well-defined semantics and that used a library of available motions. Arya employed a functional approach to designing animations in which simple keyframe animations were designed using the functional programming language ML[1]. The system provided pictures as an atomic data type, a compact set of primitive operations on the picture data type, and higher-order functions to provide more sophisticated operations for specific applications.

6 Future Work

The TANGO system provides a working prototype for the concepts in our animation paradigm. As touched upon earlier, we would like to produce further implementations and extensions of these ideas also.

One future direction may be to cast our framework in a more rigorous object-oriented environment such as C++. In TANGO we chose not to utilize the object-based appearance of the framework for local, pragmatic reasons: very few of our potential animation designers had experience with C++.

A possible object-oriented implementation might make transitions the methods that are sent to image objects. For example, we could send the move method to a particular image. Another possibility might create objects such as an "image exchange." This object would inherit default attributes such as the images' exchange motions that could be overridden as desired. Yet a further possibility is a Lisp implementation, simply due to the dynamic nature of the system and the prevalence of list structures throughout the path and transition data types.

Another future direction we plan to pursue is an implementation of a formal, stand-alone animation language with its own specialized syntax. Choosing C as a backbone environment was a practical decision influenced by our goal of producing a quality algorithm animation system. This initial experience now lays the foundation for creating a comprehensive desktop animation language definition.

Acknowledgments

I would like to thank Steve Reiss, my thesis advisor, for all his help throughout the development of this animation framework. Jeff Vitter, Paris Kaelleakis, Andy van Dam, Marc Brown, and Rob Rubin also have supplied insightful comments and suggestions. Scott Meyers read an early draft of this paper and provided helpful feedback.

References