The Generation of Animated Sequences from State Transition Systems

A.N.Clark and I.J.Palmer
Department of Computing, University of Bradford
Bradford, West Yorkshire, BD7 1DP, UK
e-mail: a.n.clark@comp.brad.ac.uk, i.j.palmer@comp.brad.ac.uk
September 18, 1997

1 Abstract

We address the issue of the construction of a computer animation system by composing together a collection of behaviours for the individual objects which are to be animated. The behaviours are developed using a formal notation for state transition systems. The notation is particularly flexible, allowing both a constructive and constraining aproach to behaviour representation. The behaviour necessary for animating Newton's Cradle for an arbitrary number of spheres is developed as an example of the method.

2 Introduction

The use of computers to assist in the production of animation sequences is well established. From simply being used to generate 'in-betweened' 2-d images for key-framed systems, to running complex simulations of particle systems, computers are used in every kind of animation today. Their use is, however, driven by the end result. The underlying method used for the animation is generally ignored in favour of assessing the aesthetic quality or the perceived 'realism' of the finished animation. This is, of course, the way in which all animations must ultimately be judged, but as animation systems and techniques become more complex, so it becomes more important to analyse the methods and approaches used to generate the animation to ensure that they are efficient and relevant. In this paper we discuss existing techniques and propose a new more formal approach to the definition of animations.

The structure of the paper is as follows. In the rest of this section we discuss existing animation techniques and identify the motivation for this work. Section 3 gives an overview of the approach. Section 4 uses the approach to construct a behaviour for a simple example. Finally, Section 5 analyses the approach and identifies directions for future work.

2.1 Traditional techniques

The oldest way in which computers can be used to assist animation production is the key-frame method. In this technique, the positions of objects are defined in certain 'key-frames', and then suitable software can be used to generate the interpolated frames in between those key-frames. Sophisticated techniques may be used to control the type of interpolation to give the desired motion [5, 12, 17, 18]. This type of system is common in 2-d animation systems, but is less common in 3-d systems. An extension of the technique, called *parametric* key-framing, is based on defining the parameters of objects in the animation at key-frames (e.g. position, velocity and acceleration) [11, 13]. This is more suited to 3-d animation, since typically it allows closer control of the many degrees of freedom present in the system. This type of animation can be seen as offering *explicit* control, since exact

Figure 1: Existing animation techniques

control can be exerted over all aspects of the objects in the animation, but limits flexibility and efficiency since exact data needs to known/calculated in advance to generate a given sequence.

A relatively recent style of animation is that which uses simulation techniques [1, 4, 3, 6, 7, 14, 16, 20]. This operates by simulating the way in the objects in the system behave (or are desired to behave) in the real world to produce their behaviour in the animation. Typical examples would be the simulation of Newton's laws of motion to produce realistic acceleration of bodies in a system. This can produce extremely convincing control of objects, but limits the exact knowledge of their resulting behaviour. As such, these systems offer *implicit* control, since their behaviour is calculated from a set of rules or formulae that are predefined. If objects do not behave as desired, these rules and formulae must be modified and the simulation repeated.

Systems exist that offer a hybrid of these two approaches in an attempt to offer the strengths of each, i.e. realistic behaviour through simulation and exact control via key-framing [10, 19]. It can also be argued, however, that they in fact incorporate the weaknesses of each in that the simulation aspect can be unpredictable and the key-framing limits flexibility. These three processes are shown diagrammatically in Figure 1.

2.2 The need for a new approach

Both key-framed and simulation based animation systems use approaches that are not derived from an analysis of the problem domain. Key-framed approaches are derived from traditional techniques, and as such the fundamental underlying concept is one of achieving an exact behaviour without specific modelling of the entities involved. This means that the effectiveness of the final result depends primarily on the skill of the animator. An example of this is shown in in work by Lasseter where by the motion of a 'hopping' object is constrained from penetrating the floor by adjusting its path, thus preventing the anomaly without an inherent model of the process [13]. The apparent realism of the final motion is entirely due to the experience and knowledge of the creator.

Simulation methods, in contrast, require accurate models to be created of real (or predicted) behaviour before anything can be defined. This behavioural model is then either used to simulate/predict some object behaviour (in the case of scientific or engineering simulation), or is used as a tool to generate some pre-meditated animation. In the former case, the undefined nature of the output is the point of the process, i.e. it is designed to discover some unknown behaviour of the system. In the latter case, the unpredictability is an annoyance, since the desired outcome is known and the simulation is being used to aid its production. For example, consider an animation of a 'pool' table in which it is desired that a ball is 'potted'. To generate this using simulation methods it is possible to create a model of the pool balls that obey Newton's laws of motion and then define the balls positions and initial velocities such that a ball is potted. This requires some pre-simulation calculations, and possibly some iterative process to achieve the desired result, but the required skill of the animator is reduced at the expense of the effort exerted in the creation of the modelling process.

The increase in complexity and required realism of animations means that more and more time needs to be spent on the creation process. This is either spent during the specification of the key-frame/interpolation information or on the development of the simulation model. A more efficient approach would be to merely define *exactly* the behaviour of the objects to produce the desired animation. This is in contrast to the key-framed approach, where the definitions are not behavioural, and the simulation approach where the definitions are behavioural but are not aimed at specifying deterministic actions. The method that we introduce in this paper is such a technique. It is a specification method that converges on the desired animated sequence from the initial set of all possible behaviours.

2.3 The state transition approach

To develop a more efficient method of producing animations, it is necessary to analyse the process from the beginning. Generally, the animation process begins with a pre-meditated notion of what the final sequence will be like. This would typically be missing some of the details of the sequence, and would be primarily a conceptual model of the finished animation. From this, some process must be used to generate what is ultimately a sequence of still images shown in quick succession. Each still image must be different from the last in such a way that their rapid sequential viewing produces the desired illusion of continuous changes in the sequence, e.g. motion of the objects in the scene.

Because each still image differs from the last by some (usually small) finite amount, it is logical and efficient to exploit this in some way. The first way that this can be achieved is by maintaining the same scene model from one image to the next. This would typically be in the form of a collection of objects linked together in a tree-type hierarchy [9], preserving geometric and visual appearance information from one frame to the next. Using this approach, the definition of the initial frame involves the creation of the objects in the scene and placing them at their initial positions. The generation of the rest of the sequence can then follow.

From the first frame, there are an infinite number of possible sequences that can follow, with only a finite set (ideally with only one member) representing the desired solution. The animation process must generate a member of the solution set in the most efficient way possible. The key-frame method works by knowing the final solution and working backwards to the initial frame. The simulation method works by generating the frames incrementally, with iteration to change the sequence if the end product is not a member of the solution set. The method we describe here takes a different approach, that of state restriction.

Initially, each object in the system is defined as an individual state machine with all possible inter-state transitions allowed, which formalises the infinite set of all possible sequences in such a way that restriction can follow. These separate object state machines can then be restricted and combined repeatedly, with the resulting system converging on the desired animation machine. Further animations can be produced by combining these complex machines to give new sequences. With this general outline in mind, we will now begin defining the process more formally.

3 Approach

Our approach to the construction of animation systems views such a system as a function:

$$\text{Behaviours} \longrightarrow \boxed{\text{Animation System}} \longrightarrow \text{Sequences of Scenes}$$

which accepts a behavioural description of a collection of objects which are to be animated, and produces a sequence of scenes which are to be displayed. The description of our approach is motivated by the following example: two perfect spheres are to be animated bouncing in two dimensions between two horizontal surfaces.

If the spheres collide they will both reverse their directions. If the animation system is viewed as a function, the input will be a description of the spheres possible behaviours and starting positions, and the output will be a sequence of sphere positions.

The animation system must produce a discrete sequence of scenes. This fact pervades our approach and we have sought to provide mechanisms which allow the behaviour of animated objects to be expressed as state transition systems. Furthermore, the ability to construct an animation from modular units which describe the behaviour of individual objects in the scene is highly desirable. Operations which transform and combine state transition systems meet this aim.

The behaviour of an object is described as a state transition system, as an expression in terms of transition system transformation and composition operations which build the object's behaviour from the behaviour of smaller sub-objects or part-behaviours. There are two alternative ways to view the composition of such an expression. Firstly, we might start with part-behaviours which are under-defined and tell only part of the story; the object is constructed by bolting the sub-parts together. We refer to this as the *constructive* approach. Secondly, we might start with part-behaviours which are over-defined, which contain the desired behaviour but which contain more besides; the object is constructed by constraining and intersecting sub-parts. We refer to this as the *constraining* approach.

Both of these approaches are useful in producing a behaviour description for an object. For example, given a behaviour description of a single sphere, the description of two spheres is constructed by merging two copies together so that the result is the sum of the parts. Alternatively, we could intersect two overlapping behaviours. The first is a description of a sphere which behaves correctly when it hits a horizontal surface and is completely unconstrained everywhere else. The second is a description of a sphere which behaves correctly unless it comes into contact with a horizontal surface where its behaviour is unconstrained. The description of a well behaved sphere is formed by merging the two behaviours so that the result is the intersection of the two parts.

The choice of whether to use a constructive approach or a constraining approach depends on the characteristics of the application. The constructive approach is useful where general statements can be made which cover large classes of legal object behaviour. The constraining approach is useful where general statements can be made which cover large classes of illegal object behaviour. We propose that in practice the specification of an object's behaviour will require a mixture of these approaches and the system which we describe in this paper supports both.

A benefit of using the constraining approach in the initial stages of object behaviour description is that in general we can start with the complete unconstrained behaviour for all object components. As such, the behaviour which we wish to exhibit is guaranteed to be present at the outset. This is not true of the constructive approach which starts with nothing and must find a way of constructing the required behaviour. Alternatively, the constraining approach presents the problem of checking when to stop since although the initial description contains the desired behaviour, it also contains illegal behaviour which must be deleted through successive constraints. The constructive approach does not suffer from this problem since it is exactly the sum of its parts and will not contain any rogue behaviours.

The behaviour of an object is represented as a state transition system [2]. For example, the unconstrained behaviour of a bouncing sphere is defined as the totally connected state transition system where the states represent the position of the sphere and the transitions represent sphere movements for a given unit of time[1].

[1]We are assuming constant sphere velocity. This can be varied to describe accelerating and decelerating spheres by increasing the complexity of the state transition system.

Since the transition system is totally connected, a sphere can move from any position to any other position in a single time unit. Starting with such a description has the advantage that the behaviour which we want to exhibit for a single sphere is complex: bouncing at walls and off other spheres must be taken into account, every angle of trajectory must be taken into account, *etc.* By starting with an unconstrained behaviour, we guarantee that all the legal behaviours are present.

Let S_1 be the unconstrained behaviour of a single sphere. To animate this behaviour, the behaviour is *executed*, from some starting position, as a non-deterministic state transition system which produces the sequence of the states which are visited. Figure 2.1 shows part of the behaviour S_1; given any starting position, a sphere may move to any other position in a single move. Notice that the sphere is not well behaved at the horizontal surfaces which are shown as dashed lines.

The animation of S_1 will, in general, produce a very unrealistic bouncing sphere. However, every once in a while, the animation will produce a correctly bouncing sphere. The behaviour S_1 is represented as the following expression $C(\Sigma)$ where Σ is a set containing all possible sphere positions.

A behaviour description is restricted by throwing away certain illegal movements. For example, we only want to allow sphere moves between adjacent positions, so we remove the possibility of the sphere jumping around at random. Notice that this leaves those moves which randomly change direction, because we will require these later when specifying the behaviour of spheres which collide and bounce of horizontal surfaces and other spheres. Such a restriction is represented as $S \backslash p$ where S is a behaviour description and p is a statement describing legal sphere moves. The result is a behaviour description in which all transitions have been removed which do not satisfy p. Restrictions may be nested, *e.g.* $S \backslash p \backslash p'$.

Let p_1 describe sphere moves which travel a constant distance, p_2 describe sphere moves which bounce correctly at the upper horizontal surface and p_3 describe sphere moves which bounce correctly at the lower horzontal surface. Part of the behaviour $S_1 \backslash p_1$ is shown in Figure 2.2 where a sphere may only move to a position which is a constant distance away from its current position. The behaviour $S_1 \backslash p_1 \backslash p_2 \backslash p_3$ describes spheres which move without jumping and which bounce correctly. Part of this behaviour is shown in Figure 2.3, where it is impossible for a sphere to move through the horizontal surfaces.

An alternative way of constraining a behaviour is to intersect it with another behaviour. The behaviour $S_1 \backslash p_1 \backslash p_2$, shown in Figure 2.4, is correct for spheres which bounce at the upper horizontal but is completely unconstrained at the lower horizontal. On the other hand, the behaviour $S_1 \backslash p_1 \backslash p_3$, shown in Figure 2.5, is correct for spheres which bounce at the lower horizontal but is completely unconstrained at the upper horizontal. The intersection of these two behaviours, shown in Figure 2.3, is correct for spheres at both horizontals. This is expressed as $S_2 = (S_1 \backslash p_1 \backslash p_2) \,\&(S_1 \backslash p_1 \backslash p_3)$. Notice that the behaviour shown in Figure 2.3 can be expressed in more than one way. This suggests that there are some useful equivalences between behaviour expressions, *e.g.* $S \backslash p \backslash p' = (S \backslash p) \,\&(S \backslash p')$, although it is beyond the scope of this paper to investigate these issues any further.

The behaviour S_2 allows spheres to randomly move sideways (zig-zag) whilst bouncing between horizontal surfaces. The complete behaviour description for a single bouncing sphere is $S_2 \backslash p_4$ where p_4 is a statement describing non-zig-zagging moves. The single-sphere system describes behaviour in which a sphere continues in the current direction until it hits a horizontal surface where it changes direction. Part of this behaviour is shown in Figure 2.6. The zig-zagging behaviour S_2 comes in useful when constructing two spheres bouncing between the horizontals since a single sphere will perform a zig-zag when it hits the second sphere.

The behaviour of a two sphere system, S_3, is described by duplicating S_2 so that each pair of sphere positions in S_2 are contained in a state of S_3. This is fig. 2.1 fig.2.2

fig. 2.3 fig.2.4

fig. 2.5 fig. 2.6

Figure 2: The development of a single bouncing sphere

fig. 3.1 3.2 3.3

Figure 3: The development of two bouncing spheres

represented as $S_3 = S_2 \times S_2$ and produces a behaviour description which is freely constructed by combining all the possible behaviours in S_2 with itself. There is no restriction on the number of times a behaviour description can be duplicated in this way, $S_2 \times S_2 \times S_2$ is the basis of a three sphere behaviour and $S_2 \times S_2 \times S_2 \times S_2$ is the basis of a four sphere behaviour, *etc.*

The sum of two behaviours adds together the moves from both and is represented as $S + S'$. Suppose that p_5 is a statement describing two-sphere system moves which never collide, p_6 is the opposite of p_5, p_7 is a statement describing two-sphere system moves which never zig-zag and p_8 is the opposite of p_7. The behaviour $S_4 = S_3 \backslash p_5 \backslash p_7$ describes only those two-sphere systems which never collide, note that there are some animations of S_4 which will terminate prematurely since the spheres may head towards each other and must stop before colliding. Once the two spheres have stopped, they will hover there forever since they are not able to collide or change course. Figure 3.1 shows part of the behaviour S_4.

The behaviour $S_5 = S_3 \backslash p_6 \backslash p_8$ describes only those two-sphere systems which bounce on collision. The restriction p_6 removes all sphere movements which do not lead immediately to or from a collision. The behaviour $S_3 \backslash p_6$ describes two sphere collisions which are completely unconstrained as to their outcome, for example they may pass through each other. The constraint p_8 forces all the collisions to zig-zag. Figure 3.2 shows part of the behaviour S_5.

The two behaviours are merged to produce the correct two-sphere behaviour description, $S_4 + S_5$. Figure 3.3 shows part of the two sphere behaviour when the spheres start at the upper horizontal, collide, bounce off each other, hit the lower horizontal and finally bounce off in opposite directions.

4 An example behaviour

This section describes a detailed worked example of using the approach described in §3. The example is Newton's Cradle in which several spheres of equivalent size and mass are suspended from a single rod by rigid string (a single example of which is shown in Figure 4). Initially, all of the spheres are at rest with their strings vertical. The system is initiated by lifting a number of spheres to the left or right and then releasing them. The system is frictionless and a number of possible behaviours are exibited depending upon how many spheres there are in total, how many are lifted at the start and how many are initially at rest. We assume that the reader is familiar with the behaviour of Newton's Cradle and that we can take the behavioural

Figure 4: A single ball from Newton's Cradle

Figure 5: Possible positions of simplified ball

specification as given.

The behaviour for the animation relies on the definitions which are given in appendix A. We also make a number of simplifying assumptions which are discussed in §5: the system is frictionless; the system does not support acceleration; the positions which spheres may occur in are -1 which corresponds to a sphere lifted to the left extreme, 0 which corresponds to a sphere at rest and 1 which corresponds to a sphere lifted to the right extreme (Figure 5); finally, spheres may have the following velocities: -1 when moving to the left, 0 when at rest, and 1 when moving to the right. For example, if a sphere is at position -1 with velocity 0 then it is at rest on the far left.

The position graph of a single-sphere system, $G_1 = C(\{0, -1, 1\})$, is totally connected as shown in Figure 6. It is impossible for the sphere to move from position -1 to position 1 without passing through position 0 so these transitions are removed, producing the graph $G_2 = G_1 \backslash p$ where $p(-1,1) = \mathit{false}$ and $p(1,-1) = \mathit{false}$ and otherwise p is *true*. The new graph is shown in Figure 7, with the deleted transitions shown as broken lines. The traces of G_2 include the following: $-1, 0, 1, 0, -1$ and $0, 1, 0, 1, 0, 1$.

The rate of change of position the sphere relates its current position to its next position, as shown in Figure 8. The rate of change graph of a single-sphere system, $G_3 = C(\{0, -1, 1\})$, is totally connected, and is shown in Figure 9. The graph which describes the free movement of a single-sphere system is $G_4 = G_2 \times G_3$. The traces of G_4 include the following:

$$(-1,1), (0,1), (1,-1), (0,-1), (-1,1)$$

and

$$(0,1), (1,-1), (1,1), (1,-1), (0,1), (1,-1)$$

The behaviour described by the single-sphere system graph is now restricted by analysing the desired behaviour of the sphere in each of the different positions. The behaviour is specified separately for each position, producing a collection of behaviour graphs. The graphs are then merged into a single graph which described the desired overall behaviour for a single-sphere system.

Firstly, the behaviour which is common to all the positions is that a sphere may continue from its current position in the direction indicated by the rate of change component. This produces the graph $G_5 = G_4 \backslash p_1$ where :

$$p_1((p_1, v_1), (p_2, v_2)) = \mathit{equal}(p_2, p_1 + v_1)$$

If the sphere is at position 0 and at rest then it may suddenly start moving due to a collision. Alternatively, if the sphere is at position 0 and is moving then it may suddenly stop or change direction due to a collision. Each of the individual possibilities is specified as a graph and then these are merged to specify the behaviour

Figure 6: Initial position graph of a single ball

Figure 7: Graph of ball with disjoint motions deleted

Figure 8: Relationship between position and rate of change of position for single ball

of the sphere at 0:

$$G_6 = G_4 \backslash (0,0) \Rightarrow (0,\_)$$

$$G_7 = G_4 \backslash (0,\_) \Rightarrow (0,0)$$

$$G_8 = G_4 \backslash (0,x) \Rightarrow (0,-x)$$

$$G_9 = G_6 + G_7 + G_8$$

G_6 describes the behaviour of single-sphere systems where the sphere starts from position 0 at rest, G_7 describes the situation where the sphere comes to rest at 0, G_8 describes the situation where the sphere changes direction at 0 and G_9 merges the graphs to produce G_9 which describes legal behaviour at 0.

If the sphere is at either extremity then it is either slowing down, at rest or returning towards 0. This is expressed as a pair of graphs for each extremity which are then merged:

$$G_{10} = G_4 \backslash (1,1) \Rightarrow (1,0)$$

$$G_{11} = G_4 \backslash (1,0) \Rightarrow (1,-1)$$

$$G_{12} = G_{10} + G_{12}$$

$$G_{13} = G_4 \backslash (-1,-1) \Rightarrow (-1,0)$$

$$G_{14} = G_4 \backslash (-1,0) \Rightarrow (-1,1)$$

$$G_{15} = G_{13} + G_{14}$$

The graph G_{10} forces the sphere to slow down at position 1 and G_{11} causes a sphere which is at rest in position 1 to fall towards 0. Graphs G_{13} and G_{14} force similar behaviour at -1. The individual behaviours are merged to produce G_{12} and G_{15} which correspond to behaviour at positions 1 and -1 respectively.

The single-sphere behaviour, G^1, is defined as

$$G^1 = G_5 \,\&\, G_9 \,\&\, G_{12} \,\&\, G_{15}$$

The traces of G^1 incude the following:

$$(-1,0),(-1,1),(0,1),(1,1),(1,0),(1,-1),(0,-1)$$

which represents a swinging sphere and

$$(-1,1),(0,1),(0,0),(0,0),(0,-1),(-1,-1),(-1,0),(-1,1)$$

which represents a sphere on the left which swings down to rest, stays at rest for two time units and then swings back up to the left.

Figure 9: Graph for rate of change of position of a single ball

Given the behaviour specification, G^1, for a single-sphere system, a behaviour specification for a two-sphere system is constructed by composing together two single-sphere systems and then imposing restrictions on the behaviour of the interactions between the spheres. The graph $G_{16} = G^1 \times G^1$ has states

$$((p_1, v_1), (p_2, v_2))$$

containing pairs of position and change in position information. The pair (p_1, v_1) describes the left hand sphere of the two-sphere system and the pair (p_2, v_2) describes the right hand sphere.

G_{16} must be restricted in two ways in order to describe legal two-sphere behaviour. Firstly, the traces of G_{16} will contain sequences of the following form:

$$((0,1),(0,-1)),((1,1),(-1,-1)),\ldots$$

which describes two spheres passing through one another in order to get to the extreme positions. Secondly, the traces of G_{16} contains sequences where a sphere will spontaneously leap up from being at rest or will suddenly come to an abrupt halt without any force being involved.

The first problem is solved by restricting G_{16} so that for each node (x_1, x_2) the information x_1 describes a sphere which is never at a position to the right of the sphere described by x_2. Since positions are integers, the restriction is performed in terms of a predicate which forces the x_1 position to be less than or equal to the x_2 position in all cases:

$$G_{17} = G_{16} \backslash ((p_1, \_), (p_2, \_)) \,\&\, p_1 \leq p_2$$

The second problem involves forcing each action in the two-sphere system to occur with respect to an equal and opposite action. For example, when a sphere on the left suddenly flies up to the left from rest, this must be in response to a sphere on the right flying in from the right and coming to rest, *i.e.* a collision occurs. The problem is solved by restricting G_{16} so that collisions are the only way in which a sphere may change direction. This is done in three stages: firstly, for any transition there must be an equal number of direction changes left and right; secondly, for each direction change in the left sphere there must be an equal an opposite direction change in the right sphere; finally, for each direction change in the right sphere there must be an equal an opposite direction change in the left sphere. In order to be useful later on, the constraint is generalised to multi-sphere systems.

The predicate *changeleft* is defined to hold when a sphere at position 0 changes direction to the left,

$$\mathit{changeleft}((p,v),(p',v')) = \mathit{equal}(p,p') \,\&\, \mathit{equal}(p,0) \,\&\, v' < v$$

The operator *changesleft* produces a list of transition pairs each of which change left. The arguments t_1 and t_2 are binary trees of the same shape and occur as graph nodes in multi-sphere systems. Each of the leaves of t_1 and t_2 are sphere states of the form (p, v):

$$\mathit{changesleft}(t_1, t_2) = (\downarrow (t_1 \times t_2)) \backslash \mathit{changeleft}$$

Similarly, the predicate *changeright* and the operator *changesright* are defined as follows:

$$\mathit{changeright}((p,v),(p',v')) = \mathit{equal}(p,p') \,\&\, \mathit{equal}(p,0) \,\&\, v < v'$$

$$\mathit{changesright}(t_1, t_2) = (\downarrow (t_1 \times t_2)) \backslash \mathit{changeright}$$

The predicate *samechanges* forces the same number of changes to occur on the right as on the left

$$samechanges(t_1, t_2) = equal(\#(changesleft(t_1, t_2)), \#(changesright(t_1, t_2)))$$

The infix predicate $\leftrightarrow$ holds between two pairs of velocities $(v_1, v_1') \leftrightarrow (v_2, v_2')$ when v_1 is the same as v_2' and v_1' is the same as v_2. The predicate represents the relationship between sphere velocities when a collision occurs at position 0.

The infix predicate $\hookleftarrow$ holds between a pair of velocities and a pair of sphere states when the sphere states are both at position 0 and the associated velocities represent a collision as defined by $\leftrightarrow$:

$$\begin{aligned}(v_1, v_1') \hookleftarrow ((p, v_2), (p', v_2')) = \quad & equal(p, p')\ \& \\ & equal(p', 0)\ \& \\ & notequal(v, v')\ \& \\ & (v_1, v_1') \leftrightarrow (v_2, v_2')\end{aligned}$$

The infix predicate $\hookrightarrow$ holds between a pair of sphere states, $((p, v), (p', v'))$, and a list of sphere states l: $((p, v), (p', v')) \hookrightarrow l$ when the left hand operand represents a change in a sphere state arising from a collision with one of the spheres in l:

$$\begin{aligned}((p, v), (p', v')) \hookrightarrow l = \ & atzero \Rightarrow exists((v, v') \hookleftarrow)l \\ where\ atzero = \quad & equal(p, p')\ \& \\ & equal(p', 0)\ \& \\ & notequal(v, v')\end{aligned}$$

The following two predicates, *causesleftright* and *causesrightleft*, hold between pairs of binary trees of sphere states, being the node values of multi-sphere systems. The first predicate holds when every change in velocity at 0 and on the left is caused by an equal and opposite change on the right. The second predicate holds when every change in velocity at 0 and on the right is caused by an equal and opposite change on the left:

$$causesleftright(t_1, t_2) = all(\hookrightarrow)(\uparrow (\downarrow (t_1 \times t_2)))$$

$$causesrightleft(t_1, t_2) = all(\hookrightarrow)(\uparrow (rev(\downarrow (t_1 \times t_2))))$$

The graph G^2 describes the legal behaviour of two-sphere systems. It is constructed by starting with the unconstrained two-sphere system behaviour G_{16} and restricting it so that all changes occur at position 0, the same number of changes always occur and each change occurs as the result of a collision:

$$G^2 = G_{17} \backslash (samechanges\ \&\ causesleftright\ \&\ causesrightleft)$$

The graph G^2 specifies the behaviour of a two-sphere system and was constructed by composing a pair of single-sphere behaviour specifications and then imposing some extra constraints. Consider two copies of G^2 which are merged so that the rightmost sphere of the first copy is overlaid on top of the leftmost sphere of the second copy. Such a merge will produce G^3 which is the behaviour specification of a three-sphere system. The behaviour specification of a four-sphere system can be constructed in exactly the same way by merging G_3 and G_2 on a common sphere. Using this technique any multi-sphere system can be specified.

The operator $\otimes$ will merge two graphs on a common element. We must be careful, however as to the order in which the restrictions are performed on the merged graphs, for example, $G^3 = G^2 \otimes G^2$ would not produce the desired behaviour specification since the restrictions which enforce that each change in velocity occurs due to a collision apply only to the individual copies of G^2 and will not apply across the system, *e.g.* between the leftmost sphere and the rightmost sphere which occur

Figure 10: Simple example animation

Figure 11: A more complex example

in different copies of G^2. In order to achieve the desired affect, the restrictions are applied *after* the merge:

$$G^3 = (G_{17} \otimes G_{17}) \backslash (\mathit{samechanges} \,\&\, \mathit{causesleftright} \,\&\, \mathit{causesrightleft})$$

and

$$G^4 = (G_{17} \otimes G_{17} \otimes G_{17}) \backslash (\mathit{samechanges} \,\&\, \mathit{causesleftright} \,\&\, \mathit{causesrightleft})$$

and in general

$$G^n = (\bigotimes_{i=2 \ldots n-1} G^i_{17}) \backslash (\mathit{samechanges} \,\&\, \mathit{causesleftright} \,\&\, \mathit{causesrightleft})$$

Frames of some animations generated in this way are shown in Figures 10 and 11. The first shows the most simple case, that of an animation starting with a single displaced ball. The second's starting position has one displaced ball at one end and two at the other. The spheres' positions were defined by the Haskell implementation of the system described in this paper, with the final sequence of images being produced using the REALISM system [15].

5 Conclusions, analysis and further work

We have identified a number of shortfalls in the mechanisms currently used for the development of behaviour in computer animation systems. We have proposed a new mechanism based upon state transition systems whereby behaviours are constructed incrementally by composing and restricting state machines which describe different elements of the overall behaviour. Behaviours can be built using a constructive approach and a constraining approach; this leads to a very flexible development method. A worked example of the approach has been given by developing a behaviour description for Newtons' Cradle with an arbitrary number of spheres. This section draws conclusions from the work, analyses the method and proposes areas for future development.

We present a critical analysis of the methods used in this paper, most of these points will be addressed in the following paragraphs on further work:

- The examples which have been presented in this paper are small and idealistic. In order to propose the method as being able to address large classes of animation applications further evidence must be presented with respect to more realistic scenarios.

- Although the system described in this paper has been implemented, the behaviour was initially rather slow due to the combinatorial nature of the representations. It must be shown that the methods which are proposed can be efficiently implemented by seeking program abstractions which can be used in a wide variety of applications.

- The method which is proposed will not apply to all animation applications. It is particularly suited to those applications which have easily identifiable components each of which has a behaviour and to those applications where the developer knows the intended behaviour in advance.
- The behavioural descriptions from existing animation systems, for example a system which models Newton's Cradle using physical laws and constraints, may not be appropriate for reuse by the methods proposed in this paper.
- As the complexity of applications grow, the method will suffer from the problem of determining whether or not the desired behaviour has been achieved and whether or not any undersirable behaviour is present.

The state transition notation which is described in this paper allows complex object behaviour to be developed for use in animation systems. Although the behaviours use a notation with a formally defined semantics, the specification of the behaviour and its verification are performed informally. If these techniques are to be used with respect to complex animation applications then verification will become increasingly important. A number of related logics are proposed for the verification of state transition systems (these are surveyed in [2]). An area for further work is to define a logic for the state transition systems used for animation and to develop a verification method.

Computer support of the development will be essential for complex animation applications and support could take a number of different forms. An automated deduction system could be used to verify a behaviour against a specification and to determine whether or not a modification leads to an incomplete or inconsistent behaviour. A program could be developed which allows behaviours to be *exercised* in various ways, allowing the developer to check that changes have the desired result. A graphical user interface could be developed which allows a developer to build up a behaviour by manipulating a visual representation of the state transition systems.

Complex animations will require far more sophisticated transition systems than those described in this paper. For example, all of the transitions which are performed in the Newton's Cradle example, assume a constant unit of time, and acceleration is not taken into account in the state information. We believe that the state machine formalism can accommodate variations which will support this type of behaviour. For example, transitions could be annotated with a length of time which it takes to move from the source state to the target state, or transition systems could be parameterised with respect to their granularity by annotating transitions with functions which generate intermediate states to any level of detail. An interesting variation is to link a number of state transition systems together by guards, the guards would detect when a particular object's state has become *well behaved* in some sense and can be animated with respect to a simpler, and therefore more efficient, state machine. An area for further work is to construct a more sophisticated animation application and to construct the desired behaviour using state transition technology. For example, a Newton's Cradle behaviour which deals with friction (possibly by extending the state components with extra decelerating components), with acceleration (possibly by extending the state components with extra acceleration components) and with a variable number of sphere positions (possibly by extending the transitions with functions which compute the intermediate states to any level of detail).

The example of Newton's Cradle has been implemented in the functional programming language Haskell [8] using the techniques which are described in this paper. Haskell is ideally suited to this task since it is *lazy*, *i.e.* it evaluates only those program expressions which are required to produce the output and never evaluates the same program expression twice. The results have been encouraging, since the programming facilities provided by Haskell allow the state transition constructs to be expressed very clearly, although some knowledge of program transformation was required in order to translate the program to exhibit acceptable performance.

References

[1] B. Arnaldi, G. Dumont, and G. Hegron. Animation of physical systems from geometric, kinematic and dynamic models. In *Modelling in Computer Graphics*. 1991.

[2] A. Arnold. *Finite Transition Systems*. Prentice Hall International Series in Computer Science, 1994.

[3] D. Baraff and A. Witkin. Global methods for simulating contacting flexible bodies. In *Computer Animation '94*, pages 1–12, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[4] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics*, 23(3):223–232, July 1989.

[5] Richard H. Bartels and Ines Hardtke. Speed adjustement for key-frame interpolation. In *Proceedings of Graphics Interface '89*, pages 14–19, June 1989.

[6] R. Barzel and A.H. Barr. A modelling system based on dynamic constraints. *Computer Graphics*, 22(4):179–188, August 1988.

[7] L. Dasheng, Y. Long, and L. Jiuchin. Robot graphics simulation based on kinematics and dynamics. In *Proceedings of Beijing International Conference on System Simulation and Scientific Computing*, pages 455–458, 1989.

[8] P. Hudak *et al.* The Haskell report. *ACM SIGPLAN Notices*, 27(5), 1992.

[9] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, second edition*. Addison-Wesley, Reading, MA, 1990.

[10] L. Forest, N. Magnenat-Thalmann, and D. Thalmann. Integrating key-frame animation and algorithmic animation of articulated bodies. In Tsiyasu L. Kunii, editor, *Advanced Computer Graphics (Proceedings of Computer Graphics Tokyo '86)*, pages 263–274. Springer-Verlag, 1986.

[11] L. Forest, D. Rambaud, N. Magnenat-Thalmann, and D. Thalmann. Keyframe-based subactors. In M. Green, editor, *Proceedings of Graphics Interface '86*, pages 213–216, May 1986.

[12] D. H. U. Kochanek and R. H. Bartels. Interpolating splines for keyframe animation. In S. MacKay, editor, *Graphics Interface '84 Proceedings*, pages 41–42, 1984.

[13] J. Lasseter. Principles of traditional animation applied to 3d computer animation. *Computer Graphics*, 21(4):35–44, July 1987.

[14] G. Miller. A dynamics-based modeler for character animation. In *Computer Animation '91*, pages 149–168, Tokyo, 1991. Springer-Verlag.

[15] I.J. Palmer and R.L. Grimsdale. REALISM: Resuable Elements for Animation using Local Integrated Simulation Models. In *Computer Animation '94*, pages 132–140, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[16] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. *Computer Graphics*, 23(3):215–222, July 1989.

[17] W. T. Reeves. Inbetweening for computer animation utilizing moving point constraints. volume 15, pages 263–269, August 1981.

[18] T.W. Sederberg and E. Greenwood. A phsyically based approach to 2-d shape blending. *Computer Graphics*, 26(2):25–34, July 1992.

[19] M. van de Panne, E. Fiume, and Z. Vranesic. Reusable motion synthesis using state-space controllers. *Computer Graphics*, 24(4), August 1990.

[20] C.W.A.M. van Overweld. An interactive approach to dynamic simulation of 3-d rigid body motions for real-time interactive computer animation. *Visual Computer*, 7(1):29–38, 1991.

A Definitions

Let $f : S \rightarrow P$ be a function with a domain $dom(f) \subseteq S$ and a range $ran(f) \subseteq P$ which are both finite sets. A function is represented as a finite collection of individual mappings $\{s_1 \mapsto p_1, s_2 \mapsto p_2, \ldots\}$ and $f(s) = p$ when $s \mapsto p \in f$. A function is total when $dom(f) = S$. A function is restricted with respect to a set $S' \subseteq dom(f)$, written $f \backslash S'$ to produce the function $\{s \mapsto p | s \mapsto p \in f, s \in S'\}$.

A graph is a 4-tuple (N, E, s, t) where N is a set of nodes, E is a set of edges, and both s and t are total functions $s : E \rightarrow N$ and $t : E \rightarrow N$. The function s maps an edge to its source node and the function t maps an edge to its target node. A graph is totally connected when, for any pair of nodes $n_1, n_2 \in N$ there exists an edge $e \in E$ such that $s(e) = n_1$ and $t(e) = n_2$. If (N_1, E_1, s_1, t_1) is a sub-graph of (N_2, E_1, s_2, t_2) then each component of the first graph is a subset of the corresponding component of the second graph.

A graph (N, E, s, t) is restricted with respect to a predicate $p : N \times N \rightarrow B$, written $(N, E, s, t) \backslash p$. The restriction is applied between the source nodes and target nodes of edges in the graph. The result is the largest sub-graph of the original for which the predicate holds between all source and target nodes of edges. The restriction is defined as follows: $(N, E, s, t) \backslash p = (N', E', s', t')$ where $E' = \{e | e \in E, p(s(e), t(e))\}$, $s' = s \backslash E'$, $t' = t \backslash E'$ and $N' = ran(s') \cup ran(t')$.

The sum of two graphs $(N_1, E_1, s_1, t_1) + (N_2, E_2, s_2, t_2)$ is the graph which contains all of the nodes and edges from both graphs. Where the two graphs have nodes in common these are merged. The result of the sum is the following graph $(N_1 \cup N_2, E_1 \cup E_2, s_1 \cup s_2, t_1 \cup t_2)$.

The intersection of two graphs $(N_1, E_1, s_1, t_1) \,\&(N_2, E_2, s_2, t_2)$ is the graph which contains all of the nodes and edges which the operand graphs have in common. The result of the intersection in the following graph $(N_1 \cap N_2, E_1 \cap E_2, s_1 \cap s_2, t_1 \cap t_2)$.

Predicates p which are used to restrict graphs using the notation $G \backslash p$ may be composed using the infix & and $\Rightarrow$ operators[2]. The conjunction operator is translated as follows: $G \backslash p_1 \,\&\, p_2$ is equivalent to $(G \backslash p_1) \,\&(G \backslash p_2)$. The implication operator ony makes sense when the graph contains nodes which are pairs: $G \backslash p_1 \Rightarrow p_2$ is equivalent to $G \backslash p$ where:

$$p(x, y) = \mathit{if}\ p_1(x)\ \mathit{then}\ p_2(y)\ \mathit{else\ true}$$

Predicates which are used to restrict graphs may be specified as patterns, where a pattern p is one of the following: an identifier i, a constant k, a wildcard _ or a

[2]Note that when not used in graph restriction predicates, these symbols have their usual meaning from positional calculus

pair of patterns (p_1, p_2). The pattern k is true only when it is applied to the value k otherwise it is false. The pattern $\_$ is true in all cases. The pattern (p_1, p_2) is true when it is applied to pairs whose first component satisfies p_2 and whose second component satisfies p_2 otherwise it is false. Identifiers in patterns allow components across different patterns to be equated, for example the predicate $(1, x) \Rightarrow (x, 1)$ is true when applied to values $((a, b), (c, d))$ where either a is not the constant 1 or both a and d are 1 and b and c are the same value.

The product of two graphs $(N_1, E_1, s_1, t_1) \times (N_2, E_2, s_2, t_2)$ is the largest graph whose nodes are pairs of nodes from N_1 and N_2, whose edges are pairs of edges from E_1 and E_2 and whose edges are only defined between pairs of nodes when the original graphs had edges between the pairwise components. The product is the graph $(N_1 \times N_2, E_1 \times E_2, s, t)$ where $s(e_1, e_2) = (s_1(e_1), s_2(e_2))$ and $t(e_1, e_2) = (t_1(e_1), t_2(e_2))$ by which we mean that s is defined only in the case that both s_1 and s_2 are defined for a given input pair and similarly for t.

A graph homomorphism is a pair of functions:

$$(f, g) : (N_1, E_1, s_1, t_1) \to (N_2, E_2, s_2, t_2)$$

which maps one graph to another graph. The signature of f is $f : N_1 \to N_2$ and the signature of g is $g : E_1 \to E_2$. The homomorphism (f, g) maps a graph to another graph by transforming the nodes with f and the edges with g whilst preserving the structure of the original graph. A homomorphism (f, g) is applied to a graph (N, E, s, t) to produce the following graph:

$$(\{f(n) | n \in N\}, \{g(e) | e \in E\}, \{g(e) \mapsto f(n) | e \mapsto n \in s\}, \{g(e) \mapsto f(n) | e \mapsto n \in t\})$$

Pairs of elements, (x, y), can be nested to produce binary trees

$$((a, b), (x, (y, z)))$$

. By traversing the trees from left to right, the tree imposes a total ordering on the values at the leaves of the tree. The operator *right* extracts the rightmost leaf value of a binary tree. The operator *left* extracts the leftmost leaf value of a binary tree. The operator *merge* is applied to two binary trees $merge(t_1, t_2)$ and produces a new binary tree by in which either the right most leaf value of t_1 has been replaced by t_2 or the leftmost leaf value of t_2 has been replaced by t_1. The outcome of *merge* is non-deterministic since either outcome will do for the purposes of this work. Given a pair of binary trees t_1 and t_2, $t_1 \times t_2$ is the binary tree which is constructed by pairing the leaves of t_1 and t_2 which occur at exactly the same position, for example

$$((a, b), (c, d)) \times ((v, w), (x, y)) = (((a, v), (b, w)), ((c, x), (d, y)))$$

If t_1 and t_2 do not have the same shape then $t_1 \times t_2$ is not defined.

A binary tree is *flattened* to produce a sequence of the values at its leaves using the operator $\downarrow$, for example $\downarrow ((a, b), (c, d)) = abcd$. A sequence s is reversed using the operator *rev* and is restricted with respect to the predicate p by $s \backslash p$. The length of a sequence is produced by the operator $\#$. Given a sequence s, the operator $\uparrow$ produces the sequence whose elements are pairs (x, s') where x is an element of s and s' is the suffix sequence of x in s, for example $\uparrow (abcd) = (a, bcd)(b, cd)(c, d)(d, \epsilon)$ where ϵ is the empty sequence. The operator *all* is applied to a predicate p, producing a new predicate which is applied to a sequence s and is true when p is true for each element of s. The operator *exists* is applied to a predicate p and produces a predicate which is applied to a sequence s and is true when p is true for at least one element of s.

Two graphs G_1 and G_2 are merged on a common element $G_1 \otimes G_2$. The outcome is a graph whose nodes are defined to be those of the product $G_1 \times G_2$ where the rightmost component of the node originating in G_1 is the same as the left-most component of the node originating in G_2. The merge is defined to produce $(merge, I)((G_1 \times G_2) \backslash p)$ where p is a predicate holding between pairs of binary trees and is defined $p(t_1, t_2) = equal(right(t_1), left(t_2))$. The symbol I is stands for the identity operator and the operators *equal*, *notequal*, $<$, $\leq$, $+$, *etc.* are defined to have the obvious meanings.

The traces of a graph is a set of all possible sequences of nodes which are encountered by starting at a particular node and then traversing the graph, choosing a single edge to traverse at each point, until a decision is made to stop (or the decision is forced because a terminal node is encountered).