A Motion Generator Framework

Martin Preston
Computer Graphics Unit, Manchester Computing,
University of Manchester, Manchester, M13 9PL, United Kingdom.
E-mail: *preston@mcc.ac.uk*
WWW: *http://info.mcc.ac.uk/CGU/staff/preston*

Abstract

There are two broad categories of research animation system; those designed to support a particular motion synthesis tool, and those whose implementation is intended to foster a particular form of motion generator design (such as object orientation).

However, neither of these approaches really provides significant assistance to the animation researcher wishing to quickly evaluate arbitrary new motion production algorithms against existing techniques. This paper addresses that need by presenting an animation system specifically designed to provide a helpful framework for a wide variety of motion generators, and discusses how these may be implemented and used.

Keywords Animation Systems, Motion Synthesis, Computer Animation

1 Introduction

The field of computer animation research has produced a variety of motion synthesis techniques, all of which are intended to assist animators in the production, maintenance, and rendering of motion in a virtual world. As a result of this research effort a number of animation systems have been developed and documented, each of which has been intended to assist the researchers in the production and evaluation of their motion synthesis approaches.

Fundamentally these *research* animation systems can be categorised into two groups. Firstly a large number of animation systems have been developed whose internal and external operations are specifically suited to the evaluation of a particular motion tool, such as Kaya [24], which presents a pleasant interface to forward dynamics, and Mira [12] which assists the animator in controlling motion through scripting.

The second category of animation system is intended to assist the programmer in a particular form of implementation or design of their algorithm [26]. For example, many researchers have attempted to derive object class hierarchies which assist the production of particular motion synthesis families [22][10], and have used these to produce full animation systems.

Ideally the purpose of an animation research system is to help in the development and exploitation of *any* new motion synthesis algorithms which may be developed. Unfortunately both of these two classes of animation environments fail to provide adequate assistance to this. The first group (the *ad-hoc* systems) are extremely good at their intended purpose, namely allowing the researcher to work with their new algorithm, but their design often does not accommodate different motion synthesis styles, and so a researcher must switch systems to develop new work.

The second style of system obstructs by its insistence on the researcher using a particular form of implementation or design, which may not be suitable for the new algorithm. This is because, whilst the system designer can anticipate some needs, he or she cannot provide a class library which will assist in all new cases, and so the hierarchy tends to require considerable effort to learn and rework.

This paper presents the design and implementation of an animation system, *PlayPen* [1], which attempts to take the best of the two approaches, and which is focused on providing as much help as possible to the animation researcher in their production and evaluation of new motion synthesis algorithms. It does this by identifying exactly what the animation researcher is interested in producing (i.e., the requirements and functionality of each motion synthesisor) and what level of help is required without the help imposing any design decisions on the researcher. In particular, whilst the animation system itself employs an object based design, the system does not require any degree of object orientation in the motion generator code.

The rest of this paper is structured as follows. Section 2 outlines the significant animation systems which fall into the two categories outlined above, and identifies those features of the two groups which are most desirable in our hybrid approach. Following this, section 3 clarifies exactly what our animation system is intended to assist in the production of, namely motion generators.

Section 4 describes the functionality *PlayPen* expects of each motion generator, and how it achieves this, in the form of an abstracted class description.

Any animation system must provide a number of features which the researcher has little interest in developing, but

1.The name of which is intended to reinforce its test-bed nature.

without which no work is possible. Section 5 describes how such facilities (rendering, recording etc.) are provided by *PlayPen*.

Section 6 contains a description of how various motion generators are integrated into the system, using these as examples of how a researcher might use *PlayPen* to quickly add new techniques. Following this Section 7 illustrates how these techniques can be used, together, in the production of motion. Finally, in Section 8, we draw conclusions as to the success of the design and areas for future investigation.

2 Related Work

As detailed earlier research animation systems can be broadly split into 2 groups, those specific to a particular motion synthesis approach (such as Virya/Kaya [24], Mira [12] and MoSys [21]) and those that employ an object oriented approach to developing a sensible class hierarchy which supports the implementation of existing motion synthesis algorithms [22][26][20].

Object orientation seems to provide considerable advantages where the problem is clearly defined, such as representation of scenery and lighting, but motion synthesis is a more complicated problem. Existing systems therefore, tend to be focused on assisting well known techniques, such as keyframing [7], and often are not suitable for synthesis algorithms they were not specifically designed for.

Two significant systems have so far been proposed which attempt to straddle the boundary between the two approaches, and provide some assistance for new techniques. The first of these, the Clockworks [5][9] system, operates by splitting the motion synthesis code into a number of autonomous elements, whose only cooperation is achieved through message passing. This system, which has been used in the development of a number of animation techniques [6], has the potential to allow the designer to exploit distributed computing, but unfortunately imposes considerable strains on the developer.

As all the interaction with the system and user is through message passing, the programmer must expend considerable effort in ensuring this interface works reliably and efficiently. However, Clockworks does succeed in at least providing a very large degree of flexibility in the style of algorithms it can support, as it is perfectly possible to allow techniques to collaborate in the production of an animation.

The Magic Theatre [1] splits the motion synthesis code into separate chunks, but attempts to allow the collaboration to take place between techniques through the use of dataflow. This approach had also been used, with considerable success, in Pixar's Menv [13] animation system. It operates by strictly describing the order in which operations will take place, with this ordering dependent on the requirements of each stage. So if one technique, which controls actor A, requires the position of a actor B to perform its work, then actor B's motion is produced first.

The Magic Theatre goes to some lengths to allow further abstraction, by adding parametric controllers, which might have been used in a form of visual animation programming (a technique visualisation systems have found to be of great use [23]). However, to the author's knowledge, the Magic Theatre was never implemented.

We can identify certain strengths from existing animation systems, namely object orientation for system internals and dataflow to simplify collaboration between techniques, which we can employ in a system which attempts to solve the remaining key problem: identifying clearly what a new motion generator needs to add to a system and what help is required to achieve this.

3 Motion Generators

In order to identify just what assistance an animation researcher requires, it is useful to consider recent developments in computer animation techniques. Following this in Section 3.2, we identify the fundamental design of *PlayPen*, and how this relates to existing motion synthesis tools.

3.1 Motion Synthesis

Whilst the early years of computer animation work were concerned with producing techniques which mimicked conventional cel animation, such as keyframing [7], the advancements in available computation power, and the widespread acceptance of computer graphics, have encouraged researchers to pursue more original approaches.

We can identify several popular, and often mutually exclusive, motion synthesis families including human locomotion [2], behavioural [19] & procedural control [27], spacetime interfaces [25], dynamics simulation [24] and performance, or captured, animation. Each of these techniques has their particular strengths, and weaknesses, so it is clear that no single methodology will triumph in the near future.

This means that a helpful animation system should strive to allow all these, and more, motion synthesis techniques to be used *together*, with minimal effort on the animators part.

Before considering how this may be accommodated it is also worth mentioning that motion synthesis is not the only area of computer animation research. For example, certain object definition models [4] have attracted research interest, as they offer distinct advantages in the production of animation, and so the ability to extend world description should also be an important feature of any system. Whilst this paper concentrates on fostering motion synthesis development, *PlayPen,* in common with other OO systems, also allows

```
class MotionGenerator {
    // Basic Features
    get/set time range()        // Either set or return the time range during which the motion generator will affect the scene
    int initialise()            // Set yourself up so that you are capable of handling queries
    int reliable()              // Are you in a reliable state
    actors_at_t(float, Actor**) // Set state for list of actors
    accept_object(Actor *)      // Tell PlayPen whether this generator can influence an object of this type
    read/write()                // Write or Read the state to or from a file
    edit()                      // Edit the motion generator
    // Advanced Features
    prepare_for_batch()         // The motion generator is about to be asked repeatedly to set actors
    stand_down()                // The system has gone out of batch mode
    Actor ** dependency(t)      // Return actors upon which you are dependent at time t
}
```

Figure 1 A Motion Generators Functionality

development of such techniques.

3.2 Fundamentals of *PlayPen* design

To enable different motion synthesis techniques to be used in collaboration it is essential to define exactly what such a technique must do. To differentiate such techniques from more arbitrary, less highly defined tools, we refer to the form of synthesis algorithm we employ as a *motion generator*.

We can define a motion generator as a collection of related functions, which when employed cause movement in an otherwise static description of a scene. This highlights an important feature of *PlayPen*, the description of the scene is the responsibility of the system, not the motion generators. The scene, which *PlayPen* defines as a collection of objects (defined using some convenient technique, such as NURBS or polyhedra), lights & cameras, is a static description of the environment. If no motion generators are applied to any portion of it, then the objects will not move.

PlayPen is designed to allow animation researchers to quickly integrate new motion synthesis techniques into the system. To assist this *PlayPen* does not impose any hierarchy on the description of the scene as far as the motion generators are concerned. Conventional animation systems employ a rigid hierarchy of object description, and whilst this is extremely useful for some motion generation methodologies (such as parametric animation [15]) any particular format of object relationship control will always make construction of other techniques more difficult.

For example, a rigid geometric description of relationships will make the integration of dynamic constraints approaches [3] awkward, which require more advanced constraint information, and potentially the need for laxness in geometric connectivity [8].

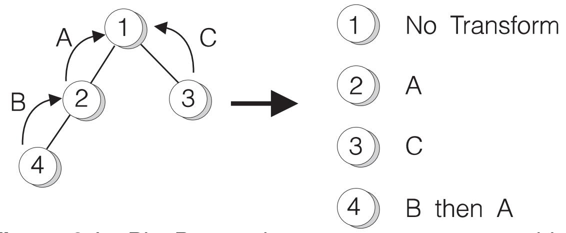

Figure 2 In *PlayPen* motion generators are not told of the scene hierarchy, though the animation system maintains it. However the relevant transformations are passed.

To this end *PlayPen* employs a two-step approach to scene control. When a scene is read in (perhaps from an existing scene description language such as MSDL [18]) the scene hierarchy is employed in the positioning of the scene, and this hierarchy is stored in case the scene needs to be exported at a later date (Figure 2). However, when a motion generator queries information about the scene each object is returned independently, with no indication of any hierarchy (a transformation matrix effectively hiding 'why' the object is in the position it is). If a motion generator wishes to employ a hierarchy *PlayPen* provides basic classes to allow an OO implementation to avoid some of the work, but *PlayPen* imposes no arbitrary decisions on how the motion generator must do this, instead the researcher is free to develop any convenient hierarchy description.

Before considering the implementation details of motion generators we must also discuss one further important feature of such entities. Our overriding goal is to allow new techniques to be used with existing, more established tools, to encourage comparison. To allow this we must allow motion generators to collaborate on the control of portions of the scene, i.e., we have to allow more than one motion generator to control one object at the same time. To implement this realistically we must employ a strict dataflow abstraction of motion control. This means that, at each time step in the animation, we must have some ordering of which motion generators are called to influence the position of each object.

In summary then, *PlayPen* is designed to assist in the construction of motion generators, which are blocks of code which when passed a portion of the scene cause the scene to be modified in some way. Furthermore these motion generators are intended to be used in collaboration with other generators, and so they should not make implicit assumptions about their level of control over the system.

4 The *PlayPen* Motion Generator Framework

PlayPen provides facilities for handling the scene (which we have already briefly discussed), rendering, saving, input/output etc. Its most important feature, though, is its ability to allow new motion generators to be integrated with it, notionally as 'plug-in' black boxes of code.

For this level of autonomy in motion generator construction it is essential that *PlayPen* defines exactly what functionality *it* requires of each generator, and this definition is the focus of this section.

Whilst *PlayPen* is constructed as an object-oriented application, it does not insist that the motion generators themselves employ this form of software engineering. To this end, though we will describe the functionality required of each motion generator in an OO style, the system could easily be modified to allow new generators to be dynamically connected (in the style made popular by visualisation systems [23]), avoiding the recompile which is required by the current version.

PlayPen expects the researcher to construct motion generators which inherit the functionality of the abstract class described in Figure 1. Provided the motion generator supports this functionality the rest of the code can be written in whichever language and style is convenient.

We will now describe the facilities *PlayPen* requires, beginning with the basic functions in Section 4.1, and then the more advanced features in 4.2 which are intended to assist in the optimization of each generator.

4.1 Basic Motion Generator Features

The fundamental function of a motion generator is to affect the position of a particular portion of the scene during some time period. It expects to be able to find out, or set (perhaps as a result of the actions of the user) the time range which the motion generator is active for. A motion generator provides this by two functions, get_range() and set_range().

The motion generator tells *PlayPen* which object it wishes to effect using a mechanism discussed in Section 5.2. Once *PlayPen* has been told that a motion generator is controlling part of the scene it will call it to modify the scene when required (the mechanism for which we will discuss in the next section). However, in order to achieve this, some motion generators may require some warning. For example, whilst simple motion generators, such as keyframers, will be able to derive the position at a particular time very quickly, some more advanced motion generators may have to perform considerable work.

Spacetime techniques [25], to pick an extreme example, may require a large amount of processing to convert their controlling information (constraint functions etc.) into positions at each point in their controlling time. *PlayPen* must be able to handle both extremes of complexity. Whilst a simple approach would be to require each motion generator to perform this 'set-up' information whenever a render was required, this may cause the system to be extremely slow.

To optimise this *PlayPen* provides two functions to allow a researcher to tune their motion generator for the system. The first function of this family, reliable(), returns TRUE or FALSE depending on whether the motion generator is in a state ready to receive queries about their positions. *PlayPen* will use this function to determine whether it can ask the motion generators to modify the scene. If reliable() returns FALSE, and if *PlayPen* requires the assistance of this motion generator (which is discussed more in Section 4.2) then it calls the initialise() function to cause the motion generator to prepare itself for later use. This function also returns TRUE or FALSE to *PlayPen*, telling it whether this call has been successful. This allows researchers to construct motion generators which can only be initialised by the user, and we shall see an example of such a generator in section 6.3.

Exactly what these two functions do internally in the motion generator will vary wildly depending on how they operate. In our spacetime example reliable() will probably return FALSE if any parameter has changed since the last initialise() was called, whereas in the keyframer reliable() will probably always return TRUE.

Once *PlayPen* knows that each motion generator is in a reliable state it will use them to determine the position of portions of the scene. It does this by passing a copy of a portion of the scene to the motion generator, and asking the generator to modify that collection of objects to correspond to the correct position in time. It does this simply by calling the actors_at_t() function, to which it passes t, which is the global time, and the collection of actors which that motion generator has effect over.

PlayPen then expects the motion generator to modify these objects before returning, using whatever technique is appropriate. Note that the construction of such a function requires the researcher to be aware of the object description system used in *PlayPen*, which is discussed further in Section 5. As *PlayPen* stores objects in a number of formats, it is therefore important that we let each motion generator inform the system which of these it is capable of dealing with.

For example, a researcher may be interested in developing a free-form deformation motion generator, which is tuned to deforming NURBS. In order to make such a tool robust *PlayPen* allows the motion generator to ensure it will only ever be passed NURBS objects. This is achieved using the accept_object() function. When *PlayPen* is presenting a list of possible subjects to a motion generator it first uses the generators accept_object() to determine whether the object could be controlled, and in this way a researcher can be sure that their code will only ever be passed objects in a format they can deal with.

PlayPen also expects some more mundane functionality, namely the ability to read or write the state of the motion generator to a file, and edit the state, through the edit() function. This is called when the user requests it, usually by selecting a button in the GUI, and is assumed to work with an interface suitable for the motion generator. Most motion generators will employ the windowed user interface, though in some cases it may be sensible to read from a command line.

4.2 More Advanced Features

This basic functionality of each motion generator provides most of the features we are likely to need in integrating a new technique. However, as performance is a concern in animation we must also provide features to enable a programmer to optimise the way their new approach is used by *PlayPen*. The motion generator class achieves this in two ways; by optimising for repeat queries and by dependency information.

PlayPen will call the actors_at_t() function for two different reasons. Firstly to force an update to the screen at an arbitrary time (perhaps to allow the user to synchronise other motion generators), and secondly as part of an actual animation sequence, when it will be repeatedly called with increasing values of t. In some motion generators we may be able to optimise the calculation performed if we know that we are in the second mode of operation, i.e., we are displaying an animation. For example we may cache useful pieces of information, or may reuse memory (thereby removing the overhead of repeatedly allocating and freeing it).

If *PlayPen* is going to start an animation sequence it calls the function prepare_for_batch() on all motion generators it knows about. Once the animation has finished it then calls stand_down() on the same set of generators.

This warning to the motion generators can help to reduce processing time if each generator is constructed with this in mind. However *PlayPen* also uses a second optimization strategy, which also allows motion generators to influence their position in the temporal dataflow system discussed in section 3.2.

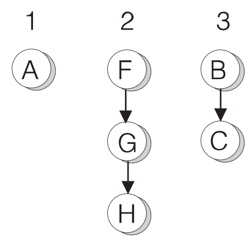

Figure 3 A dependency graph is maintained for each actor. Here actor 1 is only influenced by generator A, but actor 2 must be passed to generators F,G & H.

A motion generator is expected to modify a number of actors depending on information both internal and external to the generator. For example, a behavioural motion generator may need to examine a large set of objects, but may only modify one. As it is possible for the user to choose to only display a limited subset of the world, it is important that we decide how much calculation we must perform to maintain this view. If we can identify objects which neither appear in the view, or influence objects which are visible, then we need not use the relevant motion generators.

PlayPen achieves this by maintaining a dependency graph for motion generators and actors (Figure 3). So, for each actor at each time step, we determine an ordering of motion generators which have expressed an interest in querying those actors. *PlayPen* obtains the required information through the function dependency() that each motion generator provides. This returns the list of actors upon which this motion generators operation depends. So, in a behavioural generator, this might include all those actors which the behaviour takes into account.

Recall that it is necessary for *PlayPen* to allow more than one motion generator to control a single actor, for example an animator may wish to use two motion generators on a human mannequin to allow it to catch an object while walking. Whilst the ordering in terms of dependent actors may help us identify the ordering in which the dependency trees are evaluated, it does not always help us order the motion generators responsible for an individual actor. The desire to design a system which enables existing algorithms to work with newer ones precludes any approach which involves programmer enforced ordering, and so a compromise is used. The motion generators are stored in an ordered list within *PlayPen*, and this ordering determines the sequence in which actors are passed to the generators. This has the advantage that the ordering is clear and repeatable, and that it is easily modifiable by the animator.

This dependency graph is rebuilt for each time step, and is then searched prior to rendering, with the reliable() function

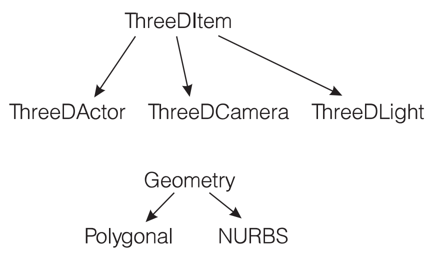

Figure 4 Class Hierarchy for ThreeDItem's

being used to determine what portions of the graph need initialising.

5 *PlayPen* Infrastructure

*PlayPen*s primary goal is to assist in the construction of motion generators, but in describing such a framework it is equally important that we outline how the rest of *PlayPen* provides appropriate support to this enterprise. This section outlines the most important features of this system, from the point of view of the researcher wishing to incorporate new algorithms within it. To this end we ignore some of the more mundane system internals whose design has no impact on such a researcher.

Section 5.1 describes the fundamentals of the system, including the actor description operated upon by the motion generator. Section 5.2 outlines the interface to this for the motion generators, and 5.3 presents the interface visible to the user.

5.1 Fundamentals

PlayPen maintains two significant lists; firstly the ordered list of motion generators and secondly a list of scenery elements. The scenery consists of a static description of all the information required to produce an image of the scene, and so includes cameras & lights as well as geometric surfaces. *PlayPen* employs an OO methodology for constructing these items, the class hierarchy for which is shown in Figure 4.

The properties of the ThreeDItem are inherited by Actors, Cameras and Lights, each of which adds additional functionality to the basic information in this abstract class. The most important piece of data encapsulated by the ThreeDItem is a 'type' field, which is supplied by whichever class inherits it. This type is used by motion generators to indicate those ThreeDItem's which it is capable of controlling, for example, a motion generator designed to allow keyframed control of lights would indicate it only accepts lights by returning the relevant type in accept_object().

The ThreeDActor adds, as well as position and orientation information, a description of its geometry, the hierarchy for which is shown in Figure 4. A generic type field is reserved for all ThreeDActors, as well as special ones for actors which employ NURBS or polygonal description. This allows the programmer to easily construct generators which can manipulate all actors, i.e., it does not manipulate the geometry.

Unlike many graphical OO decompositions each ThreeDItem does not contain a render function, instead this functionality (which is described in the following subsection) is performed by other classes which inspect the contents of these static descriptions.

5.2 Managers

PlayPen was designed to make addition of new motion generators as easy as possible, and so it is important that we reduce the degree to which a new programmer must be aware of the internal workings of the system. To this end, in order to avoid the programmer having to inspect the internal data structures of *PlayPen*, the system provides a collection of 'Manager' objects, which are simply collections of functions which provide a convenient interface to the system as a whole.

These manager objects are intended to return information in a form which is convenient for the motion generator designer, rather than *PlayPen*, and so considerable effort is expended to ensure data structures are kept which reduce many of these manager 'query' functions to simple table look-ups. *PlayPen* provides 4 of these objects

- SceneManager
- PreviewManager
- MediaManager
- ImageManager

The SceneManager is each motion generator's interface to the scene, and allows it to obtain the list of ThreeDItems for which it is and isn't responsible, as well as telling *PlayPen* that it either wishes to assume or relinquish influence over these objects. In essence the SceneManager allows the motion generator to control what work it will be called upon to perform, without the programmer having to understand the data structures of *PlayPen*. All of these functions work by passing or returning lists of ThreeDItems.

The next two managers, whose relationship is outlined in Figure 5, are responsible for hiding the complexities of input

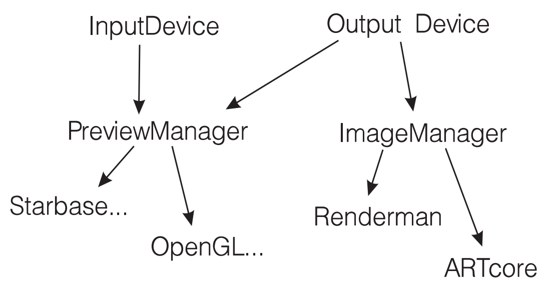

Figure 5 Input/Output Managers

and output from the researcher. Two abstract base classes are worthy of note, as it is the complexities of these that the two managers hide.

The InputDevice class allows motion generators to obtain information other than through the dialog box they create, for example to allow a user to click on the graphics screen to indicate positioning. To this end it provides three significant functions:

• int HitObject() - Return the tag of the ThreeDItem that the user has selected.

• HitPoint(float *x, float *y) - Return screen x,y that the user selected.

• CurrentPoint(float *x, float *y) - Position the user's pointer is currently over.

The OutputDevice class is responsible for taking the description of the scene in the *PlayPen* datastructures, and rendering it at a particular time, to an abstract device. It provides

• update(float t) - which renders the screen at the required time.

• update_subset(float t, ThreeDActor **) - which renders a portion of the scene.

Both of these functions cause a traversal of the scene list, and for each object the dependency graph is examined. Each motion generator in that graph is asked to modify that object before the OutputDevice, using rendering functions supplied by subclasses, inspects the actors and renders their representation.

Finally, in order to allow the InputDevice's screen based functions to work, the OutputDevice provides the function

• return_current_viewing() - which returns information about the current viewing transform.

The PreviewManager hides the interface to the real-time graphics capabilities of *PlayPen*, and attempts to allow programmers to develop code which can be ported with relative ease between platforms. It does this by implementing the abstract functionality required of the input and output devices using whichever graphics implementation is required (such as Starbase, PHIGS or OpenGL). The primitive facilities of an InputDevice go some way to providing what a researcher might require, but in order to avoid restricting development the PreviewManagers also allow access to device specific information (such as OpenGL input calls).

The ImageManager is a purely output centred manager, and allows non real-time renderers to be used in *PlayPen*. Currently the ImageManager provides output support for Renderman (in the form of an RIB file) and HP's ARTcore package. As the ImageManager is responsible for translating the information in each ThreeDItem into graphics calls, adding a new output device is relatively easy, as no render call exists within the ThreeDItem, so changes are relatively local.

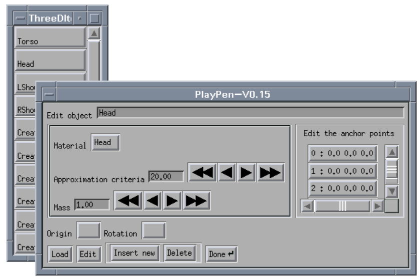

Figure 6 ThreeDItem's are presented in their own window

Finally the MediaManager (which is also output related) is responsible for handling the systems interface to other output media, such as video and film recorders. This allows motion generators to initiate recording of subsets of the scene, though in practice this manager is used more by *PlayPen* internally.

5.3 The User Interface

There are a variety of user interfaces one can envisage to the scene / motion generator system. The current implementation of *PlayPen* presents a very simple approach.

The ThreeDItem's are displayed in their own scrolling window, as shown in the background of Figure 6. Note that, as previously mentioned, any hierarchy of these objects is hidden from the user, as *PlayPen*'s philosophy is that the hierarchy is the responsibility of the motion generator. By selecting a button the user is presented with a dialog box (one of which is shown in Figure 6) allowing the editing of the static description of the ThreeDItem, in this case an actor. Where necessary this also presents geometric editing abilities via the graphics window.

Motion generators are displayed in scripts (Figure 7), which allows the user to identify the different purposes of generators which control actors and cameras (for example). The time period for which a motion generator is responsible is shown by a button in the script, and then an arrow showing the time length. If necessary a motion generator can place more buttons on the script (in the top script in Figure 7 the keyframe generator shows the temporal placing of the frames in the script). When a user selects any of the buttons associated with the generator the relevant edit() call is made.

The interface to *PlayPen* is constructed using the InterViews C++ user interface class hierarchy [11], which allows a simple correspondence between objects in the applications class hierarchy and the interfaces. However, experience with this system leads us to believe that it imposes a burden on the motion generator researcher, who then has to learn InterViews to add anything to the system. To this end work is underway to employ a Tcl/Tk interface [14], which seems to be easier to learn (at least for the simple dialogs a researcher will need to create).

6 Example Motion Generators

To demonstrate how the *PlayPen* motion generator abstraction is amenable to a wide variety of techniques this section presents four different approaches, and discusses how they fit into the system.

Section 6.1 outlines how a surface metamorphosis algorithm is implemented, 6.2 illustrates a forward dynamics approach, 6.3 a spacetime animation interface, and finally section 6.4 discusses how a procedural motion generator is integrated.

6.1 Surface Metamorphosis

The ability to modify the shape of an actor during an animation adds a large degree of flexibility to an animator, and *PlayPen* has been used to investigate mechanisms for both controlling and storing the shape of actors during an animation.

This is achieved by representing deformation as a NURBS hypersurface [16], which can be constructed by taking a number of keyframe surfaces and then producing a volume, which is later evaluated at different points in time to yield the changing shape. In *PlayPen* this is implemented as a motion generator which controls a single actor, which it insists is a NURBS (through use of types in accept_object()).

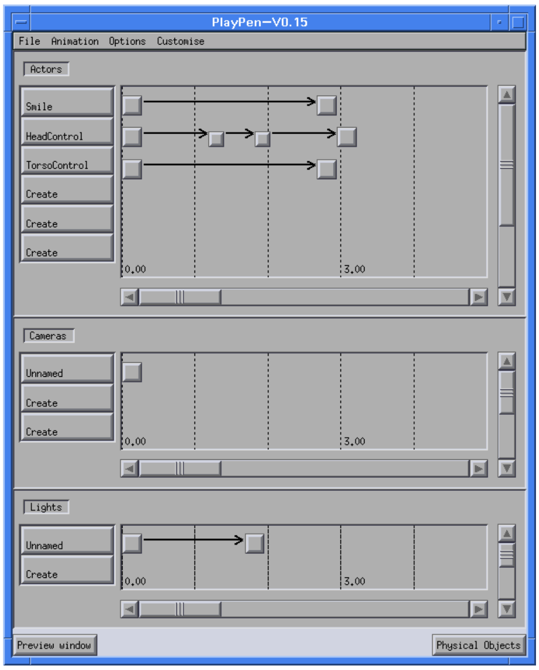

Figure 7 The motion generators are organised into scripts.

The motion generator's initialise() call takes the keyframes the user has provided, and generates the hyper-surface, so the value reliable() returns is dependent on whether the keyframes have been modified since initialise() was last called.

The evaluation of a NURBS hyper-surface is not a computationally intensive operation, but due to the data structure used requires extraction of slices through the volume. In the event of a prepare_for_batch() call these slices are extracted in advance, thereby reducing the effort required during each actors_at_t() call.

As the algorithm does not take into account the movement of any other actors the dependency() call always returns an empty list.

6.2 Forward Dynamics

Unlike the NURBS hypersurface the forward dynamics motion generator requires the ability to manipulate a hierarchy of connected actors. It achieves this by keeping a list of connections amongst actors (each connection being defined by a link point relative to the origin of the relevant actor). This system does not preclude loops occurring, and so constraint maintenance is performed by an iterative displacement approach [8].

This particular motion generator allows actors to be connected to other actors which this motion generator doesn't control (i.e., actors whose motion isn't necessarily determined by forward dynamics, or at least not *this* forward dynamics generator). To this end the dependency() call returns a list of actors upon which this generator is dependent, but which it doesn't control.

For each body a list of impulses is kept, and an initialise() call takes this list, and uses simple Euler integration to fill a table containing a sequence of positions corresponding to different times. The reliable() call therefore is dependent on whether the force list has been modified since the last initialise() was made.

As the processing involved during an actors_at_t() call is simply that of examining these pre-computed tables, no significant advantage is taken of the possibility of a warning of batch mode.

Although this implementation does not use it, it is easy to envisage a collision detection scheme based on some objects in the scene not used by this motion generator (i.e., the furniture). To implement this the dependency() call would need to return this list.

6.3 Spacetime Control

The production of motion by spacetime control [25] is one of the most computationally demanding means of animation production. *PlayPen* has been used as the basis of a technique which employs the capabilities of a parallel computer, and so demonstrates how flexible the motion generator abstraction is.

Here the *PlayPen* motion generator maintains a collection of connected objects, using similar mechanisms to that of the forward dynamics approach. Again the reliable() call returns TRUE or FALSE depending on whether the captured movement has been updated since the parameters changed. However, as the computation involved is so excessive we also take advantage of the flexibility of *PlayPen*, and produce an initialise() call which does nothing (returning FALSE).

This means that in order to change the reliability status the user has to initiate a 'set-up' in this case by selecting a button in the dialog presented by the edit() call. This enables us to prevent *PlayPen* from causing the system to hang for potentially long periods of time as the motion synthesis is performed.

As outlined in [17], this internal set up call causes a connection to be made to a parallel computer, where the actual processing is based on the information this motion generator sends to it. This means that the *PlayPen* motion generator is essentially functioning as a front end to a more complicated system.

The actors_at_t() call simply examines the precalculated positions (or splines) to generate the correct position for all the actors under its control.

6.4 Procedural Control

All of the motion generators so far described have had clearly defined dependency charts. *PlayPen* is also being used to investigate the possibilities of procedural control, in this case using a very simple language based on Tcl [14].

Here the initialise() call causes the script to be executed, and positions sampled. This script contains a number of relations, so for example the position of one actor under our control might be based on some operation performed on a number of actors which we don't directly influence. The dependency() call therefore returns this list of 'influencing' actors. The reliable() call is set depending on whether the script has been set since the last initialise was made.

7 Example Animations

In order to illustrate how *PlayPen*'s motion generators can be used in the production of an animation this section concentrates on describing the production of 2 example films. The first demonstrates how existing motion generators can be used together in the control of actors, and the second shows the ease by which new motion generators may be added to solve specific problems.

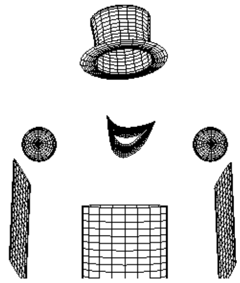

Figure 8 A stylistic figure is controlled by keyframes.

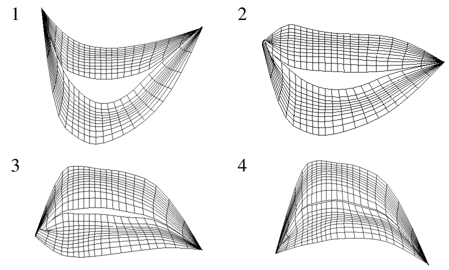

Figure 9 The smile is controlled by a hyper-surface.

7.1 'Mood Change'

This first example animation demonstrates how a researcher might combine the abilities of motion generators in the production of relatively complicated movement, in this case the control of a caricatured human figure. Here we have a model of our character which has been read from an MSDL [18] file into the scene list, and we wish to cause portions of it to move.

The bulk of the motion is purely rigid body kinematics, and so is achieved using a hierarchical key frame motion generator. First an animator creates an instance of this generator type, and then instructs the generator to assume influence over the network of bodies which compose the actor (using a dialog box supplied by the generator). Using this generator the animator can then control the bodies positions in a traditional computer animation style (the character being shown in Figure 8, and the script for this is visible in Figure 7).

However we also wish to animate the shape of the character's mouth, in order to convey a mood change (in this case from happy to sad). To achieve this we have defined our mouth as a NURBS surfaces, which we will animate using a hypersurface (as described in section 6.1, and shown in Figure 9). The animator tells this new generator to assume control over the shape of the mouth, but not its position with respect to the rest of the character.

In this way the deformation of the mouth can be controlled purely by the hypersurface motion generator. Note that, as the hypersurface generator does not manipulate the position of the mouth, it need not insist that it is called after the keyframe generator.

7.2 The Hopping Cup

The second animation demonstrates how *PlayPen* allows new motion generators to be added to solve particular problems. In this animation we employ a spacetime motion generator to generate the hopping motion of a simple two body actor (using constraints to cause the required movement).

Whilst the movement of simple rigid geometric bodies is sufficient to allow us to decide whether the movement is correct, it may not be suitable for inclusion in a larger piece of animation. We would like to improve the geometry of our actor network, and whilst we might achieve this by deforming a hypersurface, using such keyframed surfaces might obscure the physically based nature of the movement.

Consequently we write a new motion generator which is reponsible for controlling the geometry of two actors. This new generator does not change the position of the actors, but insists (using the dependency() function) that it is called *after* the spacetime generator has positioned the objects. It then examines the positions of the two actors, and places a NURBS surface over them, in this case using a model of a coffee cup. As this motion generator is produced specifically to place this cup model over these particular actors, the implementation need not be particularly general. The generator (in the actors_at_t() call) simply rotates and translates the relevant slices of the cup's control points so that they lie around the 'skeleton' actors.

This means that, as the spacetime generator causes the two actors to hop, this second generator places a NURBS surface over them, so presenting the illusion that we are using spacetime control to cause movement in deformable surfaces (as shown in Figure 10). If the animator is not happy with the movement he or she can then control it using the spacetime generator, and the cup generator will always deform the NURBS to take into account changes.

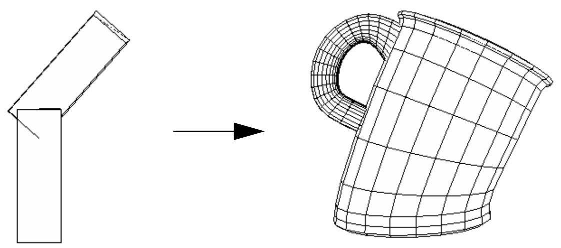

Figure 10 A motion generator takes the two body actor, and places a NURBS surface over it.

8 Conclusions and Further Work

PlayPen is intended to assist in the production & evaluation of new motion synthesis algorithms. It does this by identifying clearly the minimal requirements of these algorithms, as well as what they produce, and then building a full system around them. It does not place any restrictions on how these algorithms are either designed or implemented. This paper has demonstrated that this approach is amenable to a wide variety of techniques, and the system has proved to be a useful environment for testing these algorithms in animation production.

The most important product of this work is the identification of exactly what a motion generator requires & provides, and this allow us to pursue two further areas of investigation. Firstly it seems feasible to modify *PlayPen* to allow these motion generators to be connected dynamically (as mentioned in section 4), in order to make *PlayPen* a robust tool for both research and teaching. Secondly it enables us to examine the potential for distributing the computation required for each motion generator across a workstation cluster, using the clearly defined dependency charts to guide us in this algorithmic decomposition.

Acknowledgments

Profuse thanks to my family for their support, and the members of MC's Computer Graphics Unit (in particular Terry Hewitt & Andy Grant) for allowing me to indulge my interest in this. Dave Hutchinson gets all the credit for designing most of the objects, and the EPSRC, under whose funding some of this work was started, also deserve my gratitude!

References

[1] S. Amkraut and M. Girard. Towards an Integrated Motion Synthesis Environment. In *Computer Animation '91*, pp 131-149. Springer Verlag.

[2] N. I. Badler, C. B. Phillips and B. L. Webber. Simulating Humans : Computer Graphics Animation and Control. Oxford University Press, 1993.

[3] R. Barzel and A. H. Barr. A Modelling System Based on Dynamic Constraints. *Computer Graphics*, 22(4): 179-188. August 1988.

[4] J. Bloomenthal and K. Shoemake. Interactive Techniques for Implicit Modeling. *Computer Graphics* 24(2): 109-116. August 1990.

[5] D. E. Breen, P. Getto, A. Apodaca, D. Schmidt and B. Sarachan. The Clockworks: An Object Oriented Computer Animation System. In *Proceedings of EUROGRAPHICS '87*, pp 275-282.

[6] D. E. Breen. Choreographing Goal-Oriented Motion Using Cost Functions. In *State-of-the-art in Computer Animation* (Proceedings of Computer Animation '89), Springer-Verlag. pp 141-151

[7] N. Burtnyk and M. Wein. Computer-Generated Key Frame Animation. *Journal of the Society of Motion Picture and Television Engineers*. 80:149-153, 1971.

[8] J-D. Gascuel and M-P. Gascuel. Displacement Constraints: A New Method for Interactive Dynamic Simulation of Articulated Solids. In *3rd Eurographics Workshop on Animation & Simulation*, Cambridge, 1992.

[9] P. Getto and D.E. Breen. An Object Oriented Architecture for a Computer Animation System. *The Visual Computer*. 6(2):79-92. March 1990.

[10] E. Gobbetti and J-F. Balaguer. An Integrated Environment to Visually Construct 3D Animations. Proceedings of SIGGRAPH ‘95, 1995. In *Computer Graphics* Proceedings, Annual Conference Series.

[11] M. A. Linton *et. al.* InterViews Reference Manual. Technical report, Stanford University, 1992.

[12] N. Magnenat-Thalmann and D. Thalmann. The Use of High Level Graphical Types in the MIRA Animation System. *IEEE Computer Graphics & Applications* 3(9):9-16. November 1983.

[13] E. F. Ostby. Simplified Control of Complex Animation. In *State-of-the-art in Computer Animation* (Proceedings of Computer Animation ‘89), Springer-Verlag. pp59-67.

[14] J. K. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley, 1994.

[15] F. Parke. Parameterized Models for Facial Animation. *IEEE Computer Graphics & Applications* 2(9):61-68. November 1982

[16] M. Preston and W.T. Hewitt. Animation Using NURBS. *Computer Graphics Forum*. 13(4):229-241. October 1994.

[17] M. Preston. Parallel Spacetime Animation. In *Computer Animation and Simulation ‘95* (Proceedings of 6th Eurographics Workshop on Animation & Simulation), Springer. pp. 181-196. 1995.

[18] M. Preston, N. Gatenby and W.T. Hewitt. The Manchester Scene Description Language (MSDL V1.1). Computer Graphics Unit Technical Report CGU 88. 1995

[19] C. Reynolds. Flocks, Herds and Schools: A Distributed Behavioural Model. *Computer Graphics*, 21(4):25-34. 1987

[20] Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. Object-Oriented Modeling and Design, Chap. 19. Prentice-Hall, 1991.

[21] M. van de Panne, E. Fiume and Z. Vranesic. Reusable Motion Synthesis Using State-Space Controllers, *Computer Graphics* 24(4): 225-234, August 1990.

[22] R. Turner *et. al.* An Object-Oriented Methodology Using Dynamic Variables for Animation and Scientific Visualization. In *Proceedings of CGI’90*, pp 317-328.

[23] C. Upson *et. al.* . The Application Visualization System : A Computational Environment for Scientific Visualization. *IEEE Computer Graphics & Applications* 9(4): 30-42, July 1989.

[24] J. Wilhelms, M. Moore and R. Skinner. Dynamic Animation : Interaction and Control. *The Visual Computer*. 1988(4): 283-295

[25] A. Witkin and M. Kass. Spacetime Constraints. *Computer Graphics*, 22(4):159-168. August 1988.

[26] R. C. Zeleznik. *et. al.* An Object-Oriented Framework for the Integration of Interactive Animation Techniques. *Computer Graphics* 25(4):105-112. July 1992.

[27] D. Zeltzer. Towards an Integrated View of 3D Computer Animation, *The Visual Computer* 1(4):249-259. 1985