Parallel Spacetime Animation

Martin Preston
Computer Graphics Unit, Manchester Computing,
University of Manchester, M13 9PL, U.K.
Email: *preston@mcc.ac.uk*

Abstract. The application of parallelism to the rendering of frames for animation is a well established target for research effort. However, with the advent of more complicated motion synthesis tools, the creation of *movement*, rather than images, requires the most computation. Such techniques, whilst likely to widen the popularity of animation as a visualisation tool, are often regarded as too expensive for use in commercial motion production. This paper addresses the problem by describing a distributed algorithm for the generation of motion from high level spacetime controls, which employs parallelism to improve the ease and speed by which realistic animation can be created. Such an approach permits the use of novel architectures in animation, and widens the scope for the application of intuitive interfaces to high quality motion synthesis.
Keywords: *Parallelism, Spacetime Control, Computer Animation.*

1 Introduction

Animation has, during its widely publicised history, provided a hugely powerful tool which enables scientists and entertainers to present large quantities of information in a pleasant manner. The expense of producing a stream of images, from which a human viewer perceives motion, however, originally restricted the techniques use to the extent that the creation of high quality animation was confined to a few world-renowned centres of expertise.

Consequently, when the first graphics computers were applied to the production of animation, the focus of development was increasing the speed at which images could be generated [14]. Since the early sixties the quality and speed at which acceptable pictures may be synthesised have improved rapidly, thanks to the efforts of the computer graphics research community, many of whom have advocated the application of parallelism to frame production [12].

However, as the image production techniques advanced, the methods used by animators to create and model *motion* had remained largely similar to the techniques used by traditional cel animators. In recognition of this animation researchers have, for many years, concentrated on the development of higher level motion control techniques, which enable realistic movement to be produced *easily*, by animators who do not necessarily possess ‘cel-animation’ skills.

A variety of techniques have been generated for these purposes, all of which share a common theme, that of increased computational load in comparison to more traditional techniques. The perceived sluggishness of such tools has hampered their adoption by many animation houses, and so the state of the commercial art in animation lags considerably behind current research.

Published in *Computer Animation & Simulation (Proceedings of 6th Eurographics Workshop on Computer Animation & Simulation)*, Springer-Verlag, 1995. This copy for personal use only. © Springer-Verlag

The speed of rendering individual images for animations, then, whilst still very important, no longer restricts the widespread use of animation. Instead the most significant hindrance is the speed and ease of motion control, both for animation and interactive motion production.

This paper advocates the use of parallelism in the synthesis of *movement*, rather than purely images, as a way of addressing this problem. It describes a motion generation algorithm, employing spacetime control abstractions[16], and its efficient implementation on both tightly coupled parallel machines, and networks of connected workstations (a resource more likely to be available to a contemporary animation facility).

The adoption of a parallel approach to motion synthesis allows animators to achieve acceptable performance by increasing the number of processors available for generating the movement, which may be replayed using more conventional methods. Previously the use of parallelism in achieving such results, for general (non-particle [13]) animation, has not been widely investigated [11], and the only published attempts have achieved poor performance [5][9].

Following a discussion of the relevant spacetime control algorithms, and the limited use of parallelism in animation synthesis (in Section 2), we outline the pertinent features of the new motion generator in Section 3. In order to parallelise this approach Section 4 identifies the features of this tool which contribute to its computational expense, and from these the areas for parallel exploitation.

This technique has been used on a KSR-1/64 multicomputer, and section 5 describes the relevant details of an initial parallel design and implementation in this environment. Optimization of this basic algorithm for a variety of purposes, on both the KSR and workstations, is described in section 6. The performance of this algorithm is the topic of section 7. Finally the applicability of both this technique, and general animation synthesis tools, in a parallel setting, are discussed in section 8.

2 Related Work

Most high level motion synthesis tools rely on the application of physical properties to the virtual world. Given this information, the animator may cause motion by application of forces, the results of which are determined through the use of Newtonian simulation processes. Unfortunately, for all but the most trivial cases, such simulations are too expensive to perform interactively, and must be executed in batch runs.

In recognition of this Armstrong,Green & Lake [1], as part of their development of an efficient method of simulating the behaviour of chains of linked bodies, proposed the adoption of some parallel computation. Unfortunately this promising idea was never pursued. Friedmann & Pentland [5] describe a similar distributed implementation of a simulation process, and some parallelism was found, but the implementation used an unsuitable platform, which produced poor results. However, whatever the success of improving an implementation of dynamics simulation, most animators find force control of any animation other than simple reactive motion counter-intuitive, and so higher level interfaces to the simulation process have been developed.

The most general, and therefore promising, techniques rely on animators describing the properties of the motion most desired in statespace terms, commonly called spacetime constraints, which the motion generator uses to produce an optimal solution to the problem. If the motion is incorrect, the animator alters the hints and re-synthesises the motion.

Witkin & Kass' [16] spacetime tool operated by being supplied an initial animation path, and a collection of minimisation functions, which were used to iteratively perturb the initial trajectory in the pursuit of a 'better' path. Creation of more suitable methods of control has attracted some effort, notably from Cohen [3] and Liu *et. al.* [8], who have proposed the animator should steer the refinement process, which can be made more efficient by using advanced spline representations.

Possibly the most serious problem with spacetime control methods, though, is the computational expense of statespace minimisation. The principal strength of the spacetime approach is its intuitiveness in comparison to conventional dynamics control, but it is difficult for an animator to control a technique which requires hours of computation.

The expense of this minimisation could be ameliorated by reducing the number of times it must be performed, and Van de Panne, Fiume and Vranesic [15] have proposed creating *reusable* controllers using spacetime control. However, the creation of sufficient numbers of these controllers is often more expensive than refining a new path each time for realistic motion, and so does not provide a general solution.

In recognition of this Ngo and Marks [9] advocated the use of genetic algorithms to *evolve* animation controllers in parallel on a Connection Machine 2. This motion generator takes simple indications of the forms of motion, e.g., move left to right, and creates a controller by examining a large set of possible solutions, the desirable elements of which are combined using genetic programming techniques. The authors describe synthesis times of between 30 minutes and an hour for quite simple abstracted problems on > 4000 (1 bit) processors. Though, unfortunately, this control mechanism exhibits limitations which prevent its widespread use, and is extremely slow, it does indicate the susceptibility of this area to parallelism.

Spacetime control, then, is recognised as a promising control methodology for general animation production. Unfortunately, due both to limitations of interfaces and the large amounts of computation required, it has yet to be accepted as a commercially viable procedure. However, an implementation of this technique which can operate promptly is likely to attract widespread use. The application of parallelism to the synthesis of movement, for general problems, has not been adequately explored [11], yet seems to provide a productive solution to the problems exhibited by spacetime techniques.

3 Discretised Spacetime Control

Existing spacetime motion synthesis tools are geared towards relatively efficient implementation on a serial platform. In order to produce an algorithm which may operate in a parallel environment, and take full advantage of large numbers of processors (i.e., be scalable), it is necessary to substantially modify these approaches. This section presents a spacetime motion creation technique which is better suited to distributed computation, and supports control by a higher level interface. For the purposes of this paper only the characteristics of the algorithm which influence its parallelism are presented. The full algorithm is discussed elsewhere [10].

Section 3.1 outlines the principal features of the new approach, and sections 3.2 to 3.5 discuss the key elements of its implementation.

3.1 Parallel Motion Synthesis

We can describe the behaviour of any actor during an animation as a path through an n-dimensional space, where each dimension corresponds to some pertinent value we use to represent the actor. For example, the statespace of a single non-deformable body could be minimally described in six dimensions, 3 Cartesian and 3 Euler:

$$(x,y,z,rx,ry,rz) \tag{1}$$

Consequently the production of animation can be thought of as the creation of a path through this statespace. Such a path can be modelled as a trajectory in this space if we include time, t, as an extra dimension. Although all animation tools can be described in such terms, most hide the statespace nature of the trajectory. Spacetime motion generators, though, use the statespace as the control medium.

Most spacetime techniques operate by finding an 'optimal' path through statespace. The belief that optimised motion is inherently more realistic has its roots in the thinking that most animal motion is motivated by reducing energy expenditure. Logically, if we express terms which describe the energy expended by a particular movement, then finding movement which achieves its goals, but requires less effort, is likely to lead to better animation.

Spacetime techniques commonly require an initial trajectory as a way of providing a starting point for minimisation [16][3], the control of which is provided by animator created functions.

However another approach is to discretise the statespace itself, and refine the extents of this volume. The new statespace this algorithm produces is, then, the area in which an optimal path lies. As the volume of the statespace is reduced we get closer to the actual optimal. Indications of exactly what must be minimised are still required, but an initial, possibly misleading, trajectory is no longer always needed.

Rather than expending effort to produce an accurate path which is difficult to control, then, we concentrate our efforts on being able to interactively produce approximations, which can easily be refined to create the finished animation. This refinement process is therefore an iterative one, where the algorithm is used to reduce the subset of statespace within which the optimal path must lie. How this volume is determined is the focus of the rest of this section.

We can think of the new volume of statespace as a collection of sub-volumes, through which the optimal path passes. As we know the start position of the animation, we can view the motion synthesis process as repeatedly determining approximations to the best *exit* from each sub-volume, because this will tell us which sub-volume we need to examine next.

For an approximation to the optimal path to be useful it must provide an indication of the form of motion it contains (in order to let the animator decide whether further refinement is advantageous). This chosen best exit will provide us with enough information to construct a sample animation, i.e., a path which traverses the list of best exits we generate. To illustrate this Figure 1 shows the set of volumes, V_i, which enclose the approximate optimal path, together with a sample path.

The most important part of this process is its technique for discretising statespace into sub-volumes, as a poor tactic will hamper the generation of a plausible animation. In order to understand the factors which influence the choice of discretisation we will

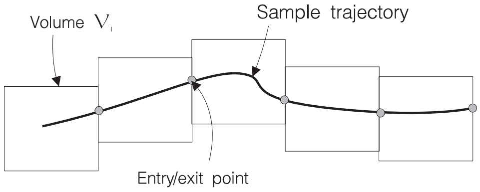


Figure 1 The path passes through a series of sub-volumes.

first discuss how an approximate best exit from each sub-volume is produced. Then, in section 3.3, we describe the discretisation algorithm used. How this technique is applied to multi-body animation is the topic of section 3.4, and finally in section 3.5 we review the strengths of the process.

3.2 Finding an exit from a sub-volume

Most spacetime techniques use energy as the focus of minimisation. However, to help the animator, we would like to hide the physical properties of the underlying world model, and so 'cost' is a more useful representation.

Given a sub-volume, centred around our current position in statespace, we wish to find the approximate best exit in a particular time. This is achieved by examining a large variety of routes, one of which is shown in Figure 2, the component costs of which can be split into

$$C(V, S_1, S_2, D) = M(V, S_2) + CP(S_2) + CR(D) \tag{2}$$

If we increase the number of routes sampled we have a greater chance of finding a more accurate exit, but obviously there is a computational overhead.

The first term of (2) accounts for the cost assigned to the movement. This is determined by a line integral over the trajectory, to which a cost may be assigned, with higher costs representing larger forces. The second term allows us to control the motion using a higher level mechanism. This works by assigning a cost to each portion of statespace, which represents the desirability of the animation traversing that point. Such control is normally achieved by allowing the user to interactively place *cost-areas* in statespace, which are regions for which a particular cost/point function allows us to find the cost influence. For example, the animator may wish to define a function with a high cost at the centre of the region, dropping to no cost at the perimeter, as a way of discouraging the trajectory from traversing this region.

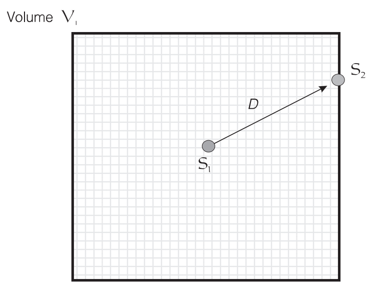
The final term, $CR(D)$, represents the cost accrued by the *route*, and allows us to account for effects such as collisions, or the cost of traversing cost-areas.

3.3 Discretising the statespace

Given the algorithm for finding the best exit from a sub-volume, we must consider how we determine the collection of sub-volumes which contain plausible animations. For the motion to be acceptable it must reach the destination in the given time. Therefore the discretisation uses two techniques to ensure acceptable routes are examined:

- A large cost-area is created which ensures that routes which approach the destination are cheaper than those which don't.
- Volumes are considered which guide the route towards the destination, though some freedom is allowed.

This second factor influences how we discretise space. We must consider volumes, whose exits we evaluate, which guide the motion towards the destination, and we must




```
follow_route(position,max_steps)
{
  choose start volume;
  cur_step = 0;
  while cur_step < max_steps
  {
    increase size of volume; cur_step++;
    exit=find_exit(volume);
    follow_route(exit,max_steps-1);
  }
  evaluate cost of this route;
}
```

Figure 2 A sample exit from a volume.

Figure 3 The motion generator algorithm.

ensure we consider as many volumes as possible whose exits actually *touch* the destination. As we are only examining a finite number of exits on each sub-volume, we must ensure that one of these points reaches the destination, which can be achieved by rotating sub-volumes so that one of the exits lies at the destination.

In order to consider plausible animations we must evaluate a wide variety of paths. This is achieved using the recursive algorithm outlined in Figure 3 where, at each point in time, we consider exiting a range of volumes, of increasing size, centred on the current point. Each chosen exit is then considered as the next step on the route. For each full route evaluated in the recursive tree, i.e., a route which has taken the maximum number of steps, we determine the total cost by adding the costs of each of the smaller movements which contribute to that path. The final approximate optimal path is then found by choosing the cheapest of all these contenders. Note that the variable `max_steps` allows us to control the resolution of the synthesised path, and so the quantity of work which must be performed.

To illustrate this process, Figure 4 shows two routes which have been evaluated as routes between A & B. A large cost-area of positive cost discourages routes which traverse it, and two paths are shown, with their component sub-volumes passing above and below. Each sub-volume is rotated so that one of its considered exits is oriented towards the destination. Note that every sub-volume is assumed to be traversed in constant time. This means that the lower route in Figure 4 takes longer to reach the destination than its counterpart.

3.4 Multi-Body Animation

An algorithm for generating a spacetime path, for a single actor, has now been

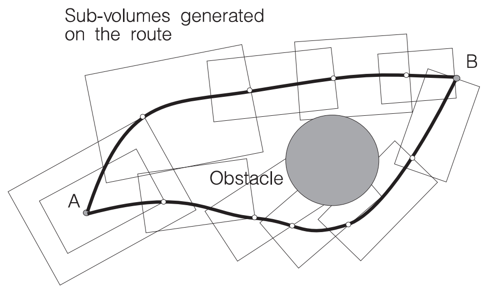


Figure 4 Two paths which are being considered,

described. However, most production animation involves the control of more complicated networks of bodies. Conventional spacetime approaches tackle this by treating the relationship between component bodies as additional dimensions in statespace, which, unfortunately, increases the synthesis times.

However we can treat each body independently, and consequently in parallel, to produce a series of statespace paths which describe the motion each component body wishes to traverse. The finished animation can then be constructed by post-processing each path to ensure connectivity is maintained. The algorithm is described in more detail elsewhere [10], and is based on previous constraint maintenance approaches [6].

Note that this approach causes the addition of a serial component to the application which cannot easily be implemented in parallel. However, this step requires little computation and, as we will see in section 7.3, will not unduly affect the scalability of the algorithm.

3.5 Algorithm Fundamentals

This motion synthesis technique has been described in abbreviated form, a more comprehensive discussion being available elsewhere [10]. In particular we have disregarded the details of finding the forces required to cause a particular motion, the production of high quality motion from the approximation (which is achieved by using these forces as impulses in a forward dynamic simulation), as well as the algorithms ability to handle collisions & control mechanisms which operate at a higher level than the placement of cost-areas.

However, none of these abilities change the fundamental algorithm, described in Figure 3, which influences the *expense* of using such a technique as a motion generation tool.

4 Parallel Properties of the Algorithm

In order to implement this synthesis technique in parallel we must determine a way of splitting the computation caused by evaluation of the algorithm (Figure 3), into a number of work units, which may be processed independently. Following an outline of the reasoning behind the chosen approach, we discuss the techniques employed to ensure the algorithm performs well using a variety of problems, in section 4.2.

4.1 Basic Parallel Design

As `follow_route` is evaluated an implicit tree is constructed, which maps out the search space that the algorithm processes. When the algorithm detects that it has taken the maximum number of steps (which have not *necessarily* brought it to the desired destination), it compares the suitability of the route taken to determine whether this is a candidate for the optimal path. When all the possible routes have been examined the cheapest is chosen as the trajectory.

At each node in the search space we are examining a particular volume around the ‘current’ point, and choosing the cheapest exit from it. As the size of this volume is dependent on the `cur_step` it is possible to characterise a node in this space by that value, and the depth of the node in the tree. A portion of the search space is shown in Figure 5, where the first figure in each node denotes the depth, and the second the magnitude of the relevant volume.

Each of these nodes corresponds to a single cycle through the loop described in Figure 3. The principal cost of this, once we disregard the cost of administration, is that of

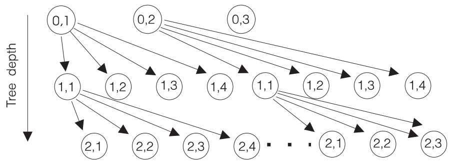


Figure 5 A portion of the search space.

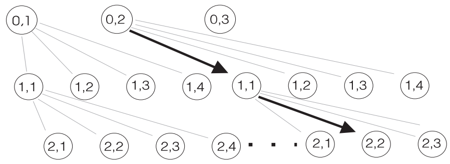


Figure 6 A path through search space.

finding the cheapest exit from a particular volume. However, as we achieve this by examining a finite number of exits, this expense is not great, and so this portion of the algorithm is not worth parallelising.

The expense of using this algorithm, however, is in the large number of sub-volumes which must be considered, which we can think of as the size of the search space. A productive method of parallelism, then, requires splitting this search space into smaller work units, and distributing these across the available processors.

It is tempting to choose a single node in the search space as our basic work unit, but there are problems with this. As we have seen each node corresponds to an iteration around the loop in Figure 3, which is intended to solve the problem “at a particular point in state-space, what is the best place to move to on the bounds of a particular volume?”.

It is important, then, to note that this is meaningless without knowing the *current* position, or the place that the preceding route has brought the actor to, and so the volume which will be considered. This preceding route (Figure 6) is determined implicitly in a serial implementation as a result of the recursive nature of the algorithm, and so means that it is impossible to treat any single node in isolation. It is for this reason that the search space is not amenable to dynamic programming techniques [2].

The expense of determining the previous path to each node, for *every* node in the search space, would be prohibitive. However, this expense can be ameliorated by considering a number of nodes, which would be treated sequentially by the serial implementation, as our work unit. In this way the preceding route only needs to be evaluated once, and so causes less of a problem. The time taken to evaluate a work unit consisting of k nodes can be expressed simply as:

$$t_{unit} = t_{setup} + (k \times t_i) \tag{3}$$

where t_i is the time taken to evaluate a single node, and t_{setup} the time required to evaluate any preceding path.

The adoption of k nodes as our unit of parallelism permits an improvement of performance over that of using a single node, though it still requires substantially more work than a serial implementation. Fortunately we can reduce this still further by recognising that for much of the time we will be processing work units which occur consecutively in the search space. If this occurs it is not necessary to evaluate the preceding path, as it will have been examined already for the previous units.

This leaves us with a simple parallelisation strategy. The search space is split into j work units, each of which consists of k nodes to be evaluated. We store the position in this space for the *start* of each work unit. These positions can then be passed to processors to enable them to efficiently reach the start of their allotted work, and evaluate the relevant portion of search space.

To allow processes to quickly reach portions of the search space the algorithm shown in Figure 3 is complicated slightly to include flags which enable a processor searching for its start position to bypass any unrequired calculation. We can achieve this because the route to the start point includes the volumes chosen at each level in the tree.

4.2 Implementation Issues

We must decide how many work units to discretise space into (j). It is tempting to choose n, where n corresponds to the number of processors available. We will never achieve linear scalability with this approach though, as the parallel implementation must perform additional work to evaluate the preceding route, but it is not likely to have a pronounced effect.This simple splitting approach, however, suffers from a more serious problem.

The cost of evaluating a particular node is governed by the expense of evaluating (2). Unfortunately the time required to evaluate this is not uniform, as the positioning of cost-areas, and portions of the scenery, alter the expense of evaluating $CP(S_2) + CR(D)$. This means that certain nodes in the search space take longer to process than others. To demonstrate this a naïve implementation was produced, and the processing times of individual nodes was accumulated. Figure 7 shows these timings, arranged sequentially to indicate that certain areas of the search space are more expensive than others.

This implementation used 10 processors, and the timings clearly show 9 peaks where the cost of evaluating the preceding route impacted on performance (the first process, of course, requires no setup). However, as expected, there are the other peaks which indicate the presence of aspects of the statespace which take longer to process. By varying the animation problem the positioning of these peaks moves, but only vanishes with the most trivial of situations.

Such a simple implementation exhibits poor scalability because the work load is not being evenly balanced across the machine, in this case the processors evaluating the 'first' portion of the search space took longer to complete their work than others. In order to improve the performance of this approach it is necessary to perform better *load balancing*. As we cannot predict where the increased areas of work are likely to lie, it is necessary to adopt a *dynamic* load balancing strategy. That is, the algorithm must be capable of determining, during processing, which processors have been allotted too much work, and so must have their load reduced.

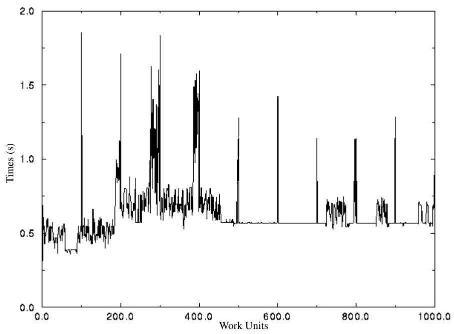


Figure 7 Timings for the different nodes.

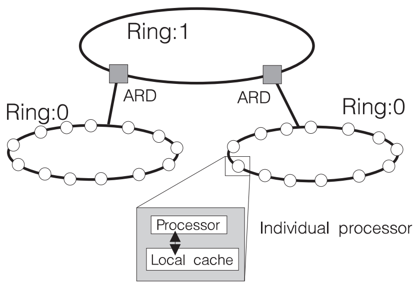


Figure 8 The KSR architecture.

5 The KSR-1 Basic Implementation

To evaluate load balancing approaches this synthesis tool was implemented on a parallel machine (the Kendall Square Research (KSR) 1) which exhibited properties which made it suitable as a test bed for algorithms which perform well on an optimal platform, but could be translated for use on more commonly available machines. Before discussing how these issues are addressed, we first outline the pertinent aspects of the architecture.

5.1 Machine Architecture

Heterogeneous parallel computers can be broadly categorised into 2 groups, those whose memory is shared between processors, and those where the memory is distributed. Unfortunately this has meant that when developing a parallel algorithm we must target one category, and be unable to employ this approach on the other group of machines.

However, recently a rival to this classification has been developed, virtual shared memory (VSM), which physically distributes the memory, but devotes considerable hardware effort into ensuring that the application running on any single processor has easy access to memory which is stored remotely. The first commercial implementation of this design is incorporated into the Kendall Square Research 1 supercomputer [4].

In the KSR-1, processors & memory units are distributed on a 2 level (0 & 1) hierarchy of rings (as shown in Figure 8). When an individual processor requires access to a portion of memory it initially looks in its own local memory/cache, but if it doesn't find it the KSR hardware (in the form of the ARD unit) fetches the memory for it. In practice, then, the memory across the entire machine is dynamic, and moves around to respond to the requirements of the applications currently executing.

From the our point of view the KSR-1 provides an ideal platform for the development of portable parallel codes, as it is capable of supporting VSM applications quickly, and so can highlight load problems, and once developed the VSM model can be emulated in more available workstation clusters.

5.2 Initial Implementation

It is common in dynamic load balanced parallel applications to employ a master/slave algorithm. Here a single process is responsible for ensuring a large number of slaves are kept occupied, which it does by doling out work from a *work list*, until the task is complete. Such a strategy is initially appealing, but for applications which are highly dynamic it has a serious flaw. If several processors finish their units of prescribed work, they must wait for the master to assign more. It takes the master some amount of time to assign a single work unit, and so several of these slaves may lie idle while waiting for jobs.

This bottleneck is caused by the physical location of the work list on a single processor, under the purview of a single process. In a distributed memory machine this is unavoidable, but in the VSM KSR-1 this is no longer the case. Given the ability to move the list around the machine, either wholly or in part, we can avoid the more serious bottlenecks.

The strategy for controlling the work performed by the motion generator operates as follows. Firstly the search space is split into some large number of work units, the number and size of which being dependent on the `max_steps` chosen by the anima-

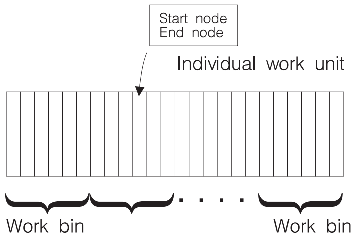


Figure 9 The work list.

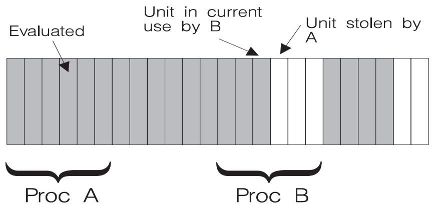


Figure 10 Stealing work units from victim processes.

tor. This list is made available to all processors as a global data structure.

The list is then split into n larger chunks of work (for n processors), which we call the *work bins* (as shown in Figure 9). A number, n, of worker processes are created, each of which evaluates portions of the search space, and is placed on its own processor. Each process refers to its work bin to find where in the search space it must begin work, and then starts processing, taking some time defined by (3). When they have each completed, they transfer back the optimal routes they have found, which the parent process chooses between by examining the costs.

6 Optimization of this Algorithm

As has been previously mentioned, a naïve implementation of such an algorithm, on any machine, is likely to perform poorly, principally due to load balancing problems. Having described the pertinent features of a KSR-1 implementation, it is now possible to address how we avoid these problems on both the target platform, and more commonly available networks of workstations.

6.1 Stealing Work

In a master/slave implementation, dynamic load balancing is performed as a result of the competitive market of slaves searching for work. A slave which is busy will concentrate on the work it has been assigned, while its colleagues will be assigned work which the busy slave would otherwise have processed. However, with a distributed work list, we cannot rely on a master distributing work fairly, and so must ensure that slaves perform the load balancing themselves, by searching the work list, and *stealing* unperformed work from other slaves' work bins (Figure 10).

This is achieved by maintaining, with the global work list, a flag which indicates whether the relevant work units are being/ have been processed yet. When a processor *begins* work on a unit it sets this flag, which prevents that unit being stolen.

Once a processor has completed the work bin it was initially allocated, it begins searching the work list. With a VSM implementation this is a cheap operation (as the machine will transfer the relevant portions of the list to local memory) but with workstation clusters this may cause expensive contention. To reduce the risk of such delays each processor traverses the list sequentially, beginning at the subsequent processors bin, and continuing in that direction, wrapping round when it reaches the end of the list. In this way we avoid the delays of several processors all beginning looking at the same place in the list.

Each worker process can be in one of two modes of operation

Victim: Here the process is working through the bin pre-allocated to it. While working, the unprocessed units within its bin are stealable, hence it is a potential victim.

Thief: Once a process can no longer find any work in its pre-allocated bin it begins looking elsewhere, using the algorithm previously described. When it finds a work unit which hasn't been used it 'marks' it as taken and begins evaluation.

Eventually all processes will enter thief mode. When each thief process can no longer find extra work it notifies its parent process, transfers the local optimal route found, and expires.

There are two principal problems with this, which we examine in the subsequent subsections, the cost of work faults, and the expense of traversing the list.

6.2 Work faults

The cost of evaluating a work unit is largely dependent on ensuring that we evaluate as many adjacent units as possible (due to the cost of evaluating the preceding route, as per (3)). However, once a thief process begins to steal work it will have to re-perform this setup, thereby causing a delay. Such a problem is analogous to the delay in an OS caused by a page fault, and is consequently termed a *work fault*.

We can avoid the worst effects of such a fault by also maintaining, along with each work bin, a *path cache* into which each processor keeps the data created during the traversal of the preceding sub-route to the current unit (i.e., the results of the work performed during each setup). Then, during each theft operation, the contents of this cache are copied along with the node information.

When the thief process begins operation it examines this cache, and reuses as much as possible, thereby reducing t_{setup} for this unit. We must allow thief processes to update the path cache, as it is possible for a considerable portion of certain bins to be evaluated by thieves, and so we would otherwise hamper the load balancing. Consequently some locking must be performed to enable multiple processors to update it correctly.

6.3 Traversing the list

On the KSR-1, thanks to the hardware support, the cost of traversing the list is low. However, on other platforms, including clusters of workstations, this expense may hamper the ability of the algorithm to balance load adequately.

This cost can be reduced, in such an environment, by increasing the granularity of the stolen units. So, rather than stealing individual units, a thief process marks a sequence of consecutive units as taken, which it would then process itself. This prevents fragmentation of the work list under extreme conditions, and enables even the slower environments to perform some load balancing.

Note that the granularity of such stolen work must be determined *experimentally* on different platforms by measuring the scalability of the implementation using animations which require extensive load-balancing. Once this size is determined it should apply to all animations, as well balanced problems will not require it, and we have tested it with the worst-case.

7 Case Studies

In order to evaluate the success of this algorithm we must examine its performance on the target platform. Whilst it is important that the algorithm produces results in acceptable time, it is equally necessary to validate that the approach is capable of using large numbers of processors, hopefully producing results faster that the minimum acceptable time.

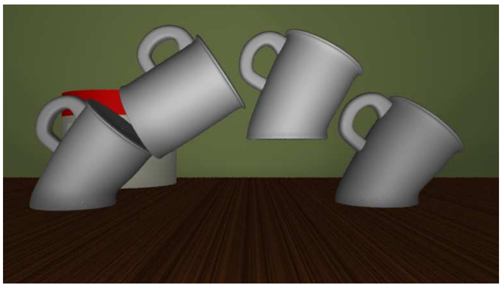


Figure 11 A two bodied actor hopping.

The parallelised algorithm, as presented, allows the animator to control the efficiency of its operation by altering 2 parameters. In order to compare times meaningfully, then, we must choose example settings. These significant parameters are

- `max_steps`, which represents the number of steps the path will be discretised into, and,
- the number of work units which the search space is split into.

For the purposes of these tests `max_steps` is set to 7, as this produces paths which are sufficiently detailed to work with, and creates a search space large enough to evaluate the effects of the number of work units, or the size of j (section 4.1). The optimal choice for this second parameter is related to the number of processors we employ, the likelihood of any load imbalances and also the speed at which we can respond to these problems.

Consequently the choice of 'ideal' granularity is closely dependent on the problem, and so we cannot determine this value in advance. However, even a poor choice will only seriously effect performance with very large numbers of processors, and so it is possible to choose an acceptable setting without too much difficulty. During these tests a granularity of 2 nodes per work unit has been chosen, but in section 7.3 the results of a poor choice are presented to justify this.

7.1 Example Animations

In order to examine the algorithms performance on different animation problems, this paper presents timings for 2 motion examples. They are

- A 2 body actor, connected by a single joint. We wish the actor to perform the hopping motion beloved of spacetime motion researchers. This motion is caused by placing a floor and obstacle in the scene, each of which are costly to enter, which thereby encourage the actor to hop as it traverses the scene. To demonstrate how spacetime techniques may be used for production animation we generate motion for a skeleton, over which a NURBS body is skinned to create an interesting character. The strobed representation of the approximated resulting coffee cups' hopping motion is shown in Figure 11.
- A 5 body representation of a mannequin, which we wish to perform a walk cycle. This requires a large number of cost-areas (15) to cause the network to appear to walk, and is one of the more complicated examples we might use this technique with. The animation produced is shown in Figure 12.

Having described the problems we now discuss the speed at which the motion may be synthesised, and the ability of the algorithm to adequately exploit large numbers of processors.

7.2 The Hopping Cup

The measured timings for the KSR-1 implementation are shown in Table 1. We can see that the time of execution quickly drops to acceptable levels (~20 seconds). Of course, if the animator finds this delay excessive he or she can reduce the `max_steps` variable to allow the route to be more quickly evaluated. We also wish to measure the ability of the algorithm to fully exploit extra processors (the speedup), so for each sample taken on *n* processors, taking t_n seconds, we plot the number of virtual processors we are fully utilising.

$$vprocs = \frac{t_1}{n} t_n \qquad (4)$$

where t_1 is the time taken on 1 processor. This graph is shown in Figure 13.

We can see that the performance of this algorithm continues to increase, though veers gradually from the ideal linear speedup (marked as the solid line). Recall that as we increase the number of processors we are actually increasing the amount of work which must be performed (due to the number of preceding routes we must process). The gentle slope away from the ideal increase indicates that this factor is having a progressively larger, but not especially critical, effect.

The kink exhibited by the graph at 50 processes is caused by the implementations' use of KSR processors which are also responsible for servicing I/O devices. It is interesting, therefore, to note that as we increase the number of processors beyond this value the load balancing strategy copes with this problem.

7.3 The Walker

The timings exhibited for this problem, and speedups measured, are shown in Table 2 and Figure 14. This problem seems to produce slightly poorer speedups, and we can

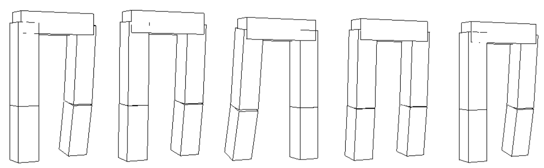


Figure 12 A walking mannequin.

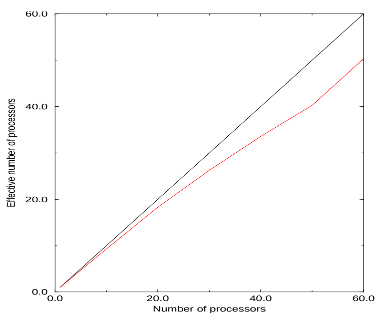


Figure 13 Speedup for the hopping cup.

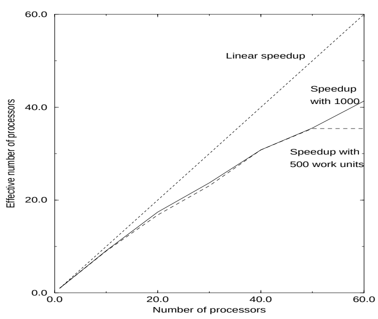


Figure 14 Speedup for the walker.

Table 1 Timings for hopping cup.

# procs	Elapsed time (s)	Efficiency(%)
1	604	100
10	65	93
20	33	91.5
30	23	87.5
40	18	83.9
50	15	80.5
60	12	83.9

Table 2 Walker Times(500 & 1000 units).

# procs	Time (500/1000)	Efficiency(%)
1	2290 / 2770	100 / 100
10	255 / 304	90 / 91
20	136 / 159	84 / 87
30	100 / 117	77 / 79
40	75 / 90	77 / 77
50	65 / 78	70.8 / 71.0
60	65 / 67	59 / 68.9

ascribe this to the increased serial processing which takes place.

Recall (section 3.4) that in order to synthesise motion for networks of actors we perform serial constraint maintenance. Whilst this requires little work in the first example, the 5 body network sometimes requires extra processing. However the effect only really becomes serious on large numbers of processors, and as we can see from Table 2, the algorithm still improves in speed rapidly, the slowdown only being noticeable once we wish to plot speedup graphs.

As an indication of the effects of a poor choice for work granularity we plot, in Figure 14, the speedup measured for both 500 and 1000 work units in our search space. Both graphs remain similar for low numbers of processors, but for large processor sets the large granularity damages the load balancing algorithms ability to work effectively, and so does not scale well beyond 50 processors. However, the difference between them, when measured in absolute time, is of the order of seconds, and so is not critical.

8 Conclusions

Recent advances in motion synthesis techniques have improved the ease by which complicated, and realistic, animation may be produced. However, the computational expense of such approaches has largely restricted their use to the academic community.

This paper has advocated the application of parallel processing to these implementations, as a way of reducing the perceived expense of high level motion control techniques. To demonstrate how this may be achieved it has described the efficient distributed implementation of such a motion generator, which uses spacetime abstractions to enable an animator to interactively create production quality motion.

From this it is possible to conclude that parallelism presents a promising solution to the difficulties of using intuitive control methodologies in animation production. Furthermore, as modern commercial animation houses often possess networks of either single or multi-processor workstations, it is likely that such an approach could attract widespread use. The development of high level motion control techniques which can exploit parallelism, then, provides a large area for future work.

Appendix

To demonstrate the speed at which motion may be synthesised in parallel, Tables 1 & 2 present the measured times for the 2 motion examples. In any practical parallel machine the timings on identical runs will fluctuate slightly, due to events out of our control, and so these timings are each averages of 5 runs. Table 2 contains timings for the two different granularities shown in Figure 14.

It is difficult to compare these times meaningfully with previous parallel motion synthesis techniques. Ngo & Marks' [9] technique employs a machine (containing huge numbers of simple single bit processors) which is especially suited to their application, and they do not provide indications of the algorithms ability to execute on varying numbers of processors, as they set out to describe a tool which it is simply convenient to run in parallel.

Friedmann & Pentland [5] do try to achieve scalability on conventional networks of workstations, but only publish results for up to 5 processors (for which they achieve only 50% efficiency), and employ a very low bandwidth network, so a straight comparison is not strictly fair.

Acknowledgments

Thanks to the students and staff of Manchesters' Computer Science department and MC's Computer Graphics Unit for their support. Some of this work was undertaken while the author was an EPSRC sponsored student, so thanks also to my supervisors Prof. F. Sumner and W.T. Hewitt.

References

[1] William W. Armstrong, Mark Green and Robert Lake. Near Real-Time Control of Human Figure Models. *IEEE Computer Graphics & Applications*, 7:39-51, June 1987.

[2] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] Michael F. Cohen. Interactive Spacetime Control for Animation. *Computer Graphics*, 26(2):293-302, July 1992.

[4] Steven Frank, Henry Burkhardt III and James Rothnie. The KSR-1 : Bridging the Gap Between Shared Memory and MPPs. In *Proceedings of Compcon '93*, February 1993.

[5] Martin Friedmann and Alex Pentland. Distributed Physical Animation. In *3rd Eurographics Workshop on Animation & Simulation*, 1992.

[6] Jean-Dominique Gascuel and Marie-Paule Gascuel. Displacement Constraints: A New Method for Interactive Dynamic Animation of Articulated Solids. In *3rd Eurographics Workshop on Animation & Simulation*, 1992.

[7] Mark Green and W.W. Armstrong. Interactive Animation Responding to the Real World. In *4th Eurographics Workshop on Animation & Simulation*, 1993.

[8] Zicheng Liu, Steven J. Gortler and Michael F. Cohen. Hierarchical Spacetime Control. In *Proceedings of SIGGRAPH '94*, pages 35-42, 1994. In *Computer Graphics* proceedings, Annual Conference Series.

[9] J. Thomas Ngo and Joe Marks. Spacetime Constraints Revisited. In *Proceedings of SIGGRAPH '93*, pages 343-350, 1993. In *Computer Graphics* proceedings, Annual Conference Series.

[10] Martin Preston. Parallel Motion Synthesis with Space-Time Control. Ph.D. Thesis, University of Manchester, 1995.

[11] Martin Preston. Parallel Motion Synthesis. In *International Workshop on HPC for Computer Graphics and Visualisation*, Swansea, 1995, To appear.

[12] F. Van Reeth and E. Flerackers. Utilizing Parallel Processing in Computer Animation. In *Computer Animation '91*, Springer-Verlag, 1991.

[13] Karl Sims. Particle Animation and Rendering Using Data Parallel Computation. *Computer Graphics*, 24(4):405-413, August 1990.

[14] Nadia Magnenat-Thalmann and Daniel Thalmann. *Computer Animation: Theory and Practice*. Springer-Verlag, Second Edition. 1990.

[15] Michiel van de Panne, Eugene Fiume and Zvonko Vranesic. Reusable Motion Synthesis using State-Space Controllers. *Computer Graphics*, 24(4):225-234, August 1990.

[16] Andrew Witkin and Michael Kass. Spacetime Constraints. *Computer Graphics*, 22(4):159-168, August 1988.