Adapting Gosper's Hashlife Algorithm for Kinematic Environments

William M Stevens

Department of Physics and Astronomy, Open University, Milton Keynes, UK, MK7 6AA

Abstract. Gosper's *hashlife* algorithm is adapted and applied to a three-dimensional kinematic environment by simulating the environment using interleaved simulation periods, each of which can be modeled using a different set of cellular automaton rules. Information about the global state of the environment is stored in order to decide when to make a transition from one set of cellular automaton rules to another. The adaptations are described in the context of a specific environment but can be applied to other similar environments.

1 Introduction

Bill Gosper's ingenious *hashlife* algorithm [1] was originally applied to Conway's Game of Life. Gosper pointed out that it could be extended to other geometries, dimensions and number of states per cell. The algorithm has recently been applied to other cellular automata (CAs) using the open source Golly CA simulator [2]. Self-replicating programmable constructors (SRPCs) that were too large to implement or simulate using the technology available at the time of their conception have recently been simulated using the *hashlife* algorithm [3–6].

Usually when a constructing machine embedded in a cellular automaton environment needs to make a new structure it does so by using CA rules that are designed to allow empty space to turn into a component part on demand: there is no "conservation of matter". One reason for this is that the CA environments that these systems are embedded in do not have any model of motion: components and machines are fixed in place.

If we introduce the constraints that component parts cannot be created or destroyed or transformed into different types of component part then we must also provide a mechanism by which component parts can be moved from one location to another.

If a constructing machine cannot create component parts on demand then it must obtain them from its environment. Either the machine must be mobile and able to seek component parts within its environment, or individual parts within the environment must move around so as to encounter the constructing machine which can then make use of them.

Additionally a constructing machine must be capable of positioning component parts correctly relative to other parts in the machine being constructed.

Either the machine being constructed must be manouverable or the constructing machine must contain a subsystem (perhaps the whole machine) which is capable of moving a component part to a specific location.

These considerations lead to a strong case for bestowing mobility not only upon component parts but also upon larger structures. Once a means of moving structures in space is provided, we are faced with the problem that a structure may fall apart when it is moved unless some means of connecting neighbouring component parts is also provided.

These are among the considerations that led to the environment described in Sect. 2.

2 A 3D Kinematic Environment with 6 Part Types

This section describes a discrete space, discrete time 3D kinematic simulation environment called CBlocks3D. This is a development of the previously published 2D CBlocks environment [7], having a greatly reduced set of component parts and not requiring any ability to create component parts out of empty space. It can be regarded as an implementation of the kinematic environment originally proposed by von Neumann [8]. There are 6 different types of part in the CBlocks3D environment: a signal propagation part (the *wire* part), a signal processing part (the *nor* part), a part for moving other parts (the *slide* part), a part for rotating other parts (the *rotate* part), a part for connecting other parts (the *fuse* part) and a part for disconnecting other parts (the *unfuse* part). These part types are chosen so as to be as simple as possible whilst spanning the operations required for construction within the environment. Boolean signals can pass between neighbouring parts. It takes one time unit for a part to respond to an input signal. In a single time unit, a part may move one unit in any one of six directions under the action of a *slide* part. When a part moves, all parts directly or indirectly connected to it also move.

2.1 Describing parts

Standard notation from set theory is used in this section. Readers unfamiliar with this notation should consult reference [9] or [10].

We define six direction vectors

$$\begin{aligned} EAST &= (1,0,0), & WEST &= (-1,0,0), \\ NORTH &= (0,1,0), & SOUTH &= (0,-1,0), \\ FRONT &= (0,0,1), & BACK &= (0,0,-1). \end{aligned}$$

D denotes the set of these vectors

$$D = \{EAST, WEST, NORTH, SOUTH, FRONT, BACK\}.$$

We define the function

$$\text{opposite}((x,y,z)) = (-x,-y,-z).$$

L is the set $\{True, False\}$, and T denotes the set of part types

$$T = \{wire, nor, slide, fuse, unfuse, rotate\}.$$

A part is completely described by the tuple

$$(P.location, P.primary, P.secondary, P.type, P.output, P.connect)$$

where

$$\begin{aligned}
P.location &\in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z},\\
P.primary &\in D,\\
P.secondary &\in D \setminus \{primary, \text{opposite}(primary)\},\\
P.type &\in T,\\
P.output &\in L \times L \times L \times L \times L \times L \text{ and}\\
P.connect &\in L \times L \times L \times L \times L \times L.
\end{aligned}$$

P.location is a 3-tuple (x, y, z) that specifies the location of the part.

Two vectors are needed to specify the orientation of a part P in three dimensional discrete space. The primary axis *P.primary* is a vector that lies on the line from the centre of P to the centre of one face of P (this face is referred to as the *active face* of P). The secondary axis *P.secondary* is perpendicular to the primary axis. For example, in Table 1 the primary axis of each part points up the page ($NORTH$) and the secondary axis points to the right of the page ($EAST$).

The notation $X[Y]$ is used to refer to the Yth element of the tuple X. It is convenient to use the direction vectors D to index the *P.output* and *P.connect* tuples, so we define that the vectors $NORTH$, $EAST$, $SOUTH$, $WEST$, $FRONT$ and $BACK$ can be used to index the 1st, 2nd, 3rd, 4th, 5th and 6th elements of a tuple respectively.

$P.connect[d] \in L$ where $d \in D$ specify the connectivity state of P. If a part P is connected in a particular direction d to a neighbouring part Q then $P.connect[d] = True$ and also $Q.connect[\text{opposite}(d)] = True$. If a part P is not connected to its neighbour Q which lies in direction d, then $P.connect[d] = Q.connect[\text{opposite}(d)] = False$. If a part P has no neighbour in direction d then $P.connect[d] = False$. When a part is moved by a *slide* part all parts connected to that part also move. A moving part P will also push a neighbouring part Q that lies in the direction of motion of P, even if Q is not connected to P. A part can be rotated by a *rotate* part regardless of its connectivity state. When a part is rotated by a *rotate* part its connectivity state remains unchanged.

Boolean signals can pass between the faces of neighbouring parts. Neighbouring parts do not need to be connected in order for signals to pass between them. Each face of a part acts either as an input or as an output. It takes one time unit for a signal to propagate from a part's inputs to its outputs or for a part to respond to signals at its inputs. When we talk of the value of a signal at an input face of a part, this is the value being ouput by an abutting face of a neighbouring part, or *False* if there is no abutting face.

$P.output[d] \in L$ where $d \in D$ are the outputs of P. So for example if we have an isolated *nor* part P with $P.primary = EAST$, the values of its outputs will be $P.output[EAST] = True$ and $P.output[d] = False$ for all other $d \in D$.

Table 1 describes the function of each type of part. A graphical representation is shown for each part type. In Table 1, the letters N, E, S, W, F and B are used to refer to the value of signals at the $NORTH$, $EAST$, $SOUTH$, $WEST$, $FRONT$ and $BACK$ inputs of a part. Any output not specified in Table 1 has the value *False*. In these diagrams $NORTH$ is up the page, $EAST$ is to the right of the page and $FRONT$ is out of the page. Note that the parts are shown here in one particular orientation and the function of each part is described with respect to this orientation. The Boolean $\neg$ (negation) and $\vee$ (OR) operators are used in Table 1.

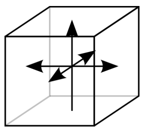
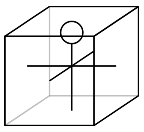
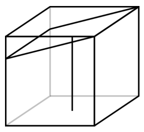
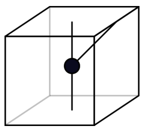
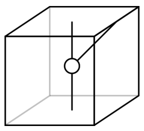
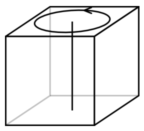
Wire	Nor	Slide
		
$output[d] := S$ for $d \in D \setminus \{SOUTH\}$	$output[NORTH] :=$ $\neg(E \vee S \vee W \vee F \vee B)$	If S then move the part that lies $NORTH$ one unit $EAST$
Fuse	Unfuse	Rotate
		
If S then connect the parts that lie $NORTH$ and $NORTH$-$EAST$	If S then disconnect the parts that lie $NORTH$ and $NORTH$-$EAST$	If S then rotate the part that lies $NORTH$ through 90 degrees

Table 1. Part types in CBlocks3D.

The *wire*, *nor* and *rotate* parts have rotational symmetry about their primary axis and can therefore be in any one of 6 functionally distinct orientations. The *slide*, *fuse* and *unfuse* parts have no rotational symmetry and can therefore be in any one of 24 distinct orientations.

All parts except the *nor* part have a single input which is at the $SOUTH$ face in Table 1. The *nor* part has 5 inputs at the $EAST$,$WEST$,$FRONT$,$BACK$ and $SOUTH$ faces in Table 1.

A self-replicating programmable constructor made from approximately $60,000$ parts was implemented in this environment [11, 12]. The SRPC collects parts from a disorganised collection in the environment and then uses those parts to construct any specified machine, with self-replication as a special case. The need to simulate this SRPC on inexpensive hardware in a matter of days rather than weeks was the primary motivating factor for developing an efficient simulation algorithm for the CBlocks3D environment.

A brief description of an important subsystem of the SRPC can be found in reference [13]. This subsystem is capable of identifying any part presented to it.

3 Simulating the CBlocks3D Environment

When the CBlocks3D environment was first devised, a simulator was developed that iterated over every part in the environment at each time step to calculate the effect that each part had on other parts. The simulator was later refined so as to simulate only those parts likely to change state from one time step to the next.

The fact that structures made from connected parts can move together in a single time step means that the environment cannot be directly implemented in a CA. In a CA the state of a cell one time step into the future is affected by cells in a finite neighbourhood. In the CBlocks3D environment a moveable structure can have any length in any dimension. As a consequence of this one end of an arbitrarily long rod can end up moving because the far end of the rod was acted upon by a *slide* part in the previous time step.

Given that the CBlocks3D environment cannot be implemented using a CA and that there is no difficulty in using other methods to simulate it, there would be no cause for further consideration of CA were it not for the existence of the *hashlife* algorithm. The potential performance improvement that the *hashlife* algorithm offers, coupled with the knowledge that several structures in the SRPC being simulated have a high degree of spatial and temporal regularity, which *ought* to be capable of being simulated in a more efficient way, strongly motivated further consideration of whether CA techniques could be adapted to simulate the CBlocks3D environment.

Although the possibility of arbitrarily sized moving structures means that in general the environment cannot be implemented using a CA, it is perfectly possible to simulate periods of time during which no movement occurs using a CA. One possible end point of such a period is shown in Fig. 1. Here a *slide* part is activated, and via its action on part A it will cause structure B to move one unit to the right in the next time step. In the first simulator that was written for the CBlocks3D environment an algorithm was written to work out which parts in a structure move when any part in the structure is acted upon by a *slide* part. This algorithm worked by propagating movement information from one part to another in a similar way to the way that information is propagated in a CA. This led to the realisation that a separate set of CA rules can be used to propagate movement information from one part to another, and another set can be used once all movement information has been propagated to carry out the movement operation.

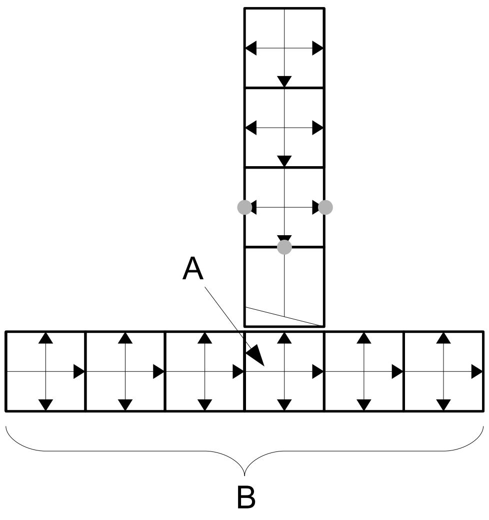


Fig. 1. Structure B is about to be moved by the *slide* part.

4 An Algorithm Based on Gosper's *Hashlife*

Firstly we devise a set of CA transition rules for carrying out all operations except movement of parts in the CBlocks3D environment. These rules will allow any simulation to be carried forward to a state where movement of parts is about to occur. This set of transition rules is referred to as the NORMAL set.

Then we devise a second set of CA transition rules that allow information about movement to propagate from one part to another. This set is referred to as the PROPAGATION set. A third set of transition rules is used to perform movement. This set is referred to as the MOVEMENT set.

After this we show how information about the global state of the environment can be used to decide when to switch from one set of rules to another. The *hashlife* algorithm is then modified to allow results calculated from all three sets of CA rules, as well as state information for subregions of the environment, to be incorporated into the same data structure.

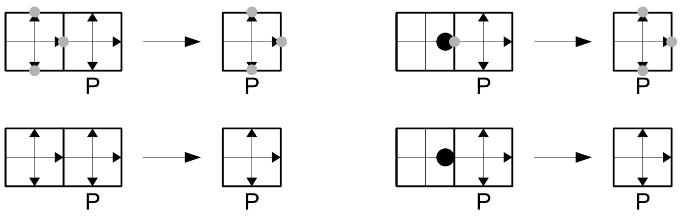
There are some possible conflicts in these rule sets: it is possible for two parts to be acted upon by both a *fuse* part and an *unfuse* part at the same time, and for two or more active *rotate* or *slide* parts to act upon the same part. Further discussion of these conflict situations and how they may be dealt with is deferred to Sect. 5.

The NORMAL set of CA rules is by far the most complex of the three. It is given as a set of If-Then rules below.

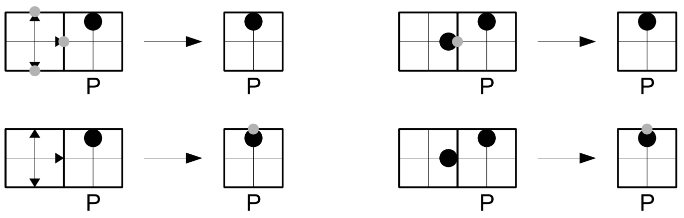
Two parts are *neighbours* when they have abutting faces. Two parts are *diagonal neighbours* when they share a single edge. Graphical examples are shown beneath each rule. The graphical examples are not intended to cover all possibilities encompassed by each rule, but rather to aid the reader in understanding the rules. Small gray circles are used to indicate *True* outputs. In the rules that deal with connectivity a small gap between parts is used to show that they are not connected. In the rule concerning the *slide* part, a grey circle with an arrow in it is used to indicate a part that is marked for movement.

In these rules P refers to a part occupying a cell. These rules show how P changes state from time t to time $t+1$. If no rule is applicable then a cell remains unchanged. Any outputs of P not explicitly mentioned in these rules are set to *False*.

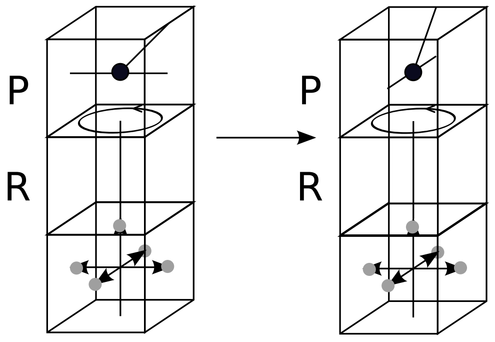
If *P.type* = *wire* and the input of P faces an output of a *wire* or *nor* part outputting a *True* signal then *P.output*$[d]$ = *True* for $d \in D\backslash\{\text{opposite}(P.primary)\}$ at $t+1$. Otherwise these outputs will be *False* at $t+1$.



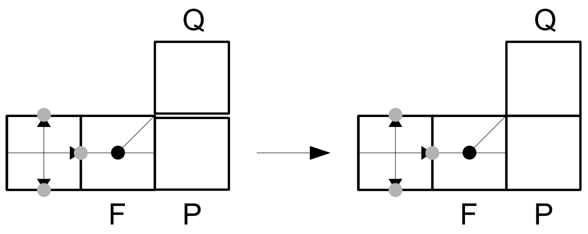
If *P.type* = *nor* and any of the inputs of P face an output of a *wire* or *nor* part outputting a *True* signal then *P.output*[*P.primary*] = *False* at $t+1$. Otherwise *P.output*[*P.primary*] = *True* at $t+1$.



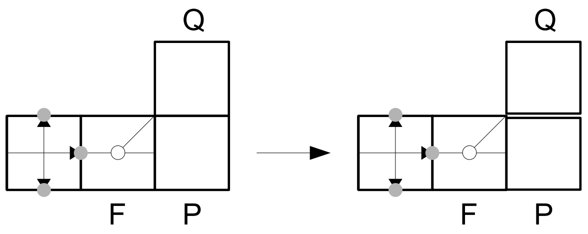
If P has a *rotate* part R as a neighbour and *R.primary* points toward P and the input of R faces an output of a *wire* or *nor* part outputting a *True* signal then P will rotate through 90 degrees anticlockwise about *R.primary* by $t+1$. Note that the state of the *P.connect* tuple is not changed by a rotation.



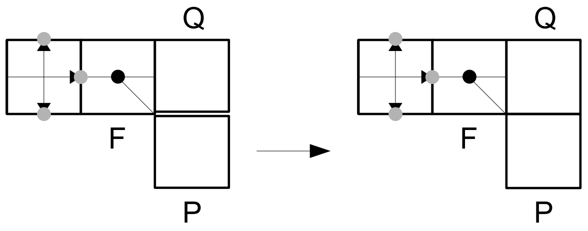
If P has a *fuse* part F as a neighbour and *F.primary* points toward P and the input of F faces an output of a *wire* or *nor* part outputting a *True* signal and P has a neighbour Q in the direction of *F.secondary* then at $t+1$ P and Q will be connected.



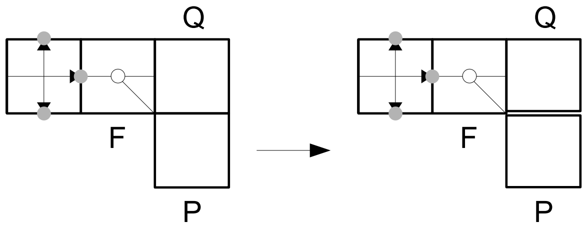
If P has an *unfuse* part F as a neighbour and *F.primary* points toward P and the input of F faces an output of a *wire* or *nor* part outputting a *True* signal and P has a neighbour Q in the direction of *F.secondary* then at $t+1$ P and Q will be disconnected.



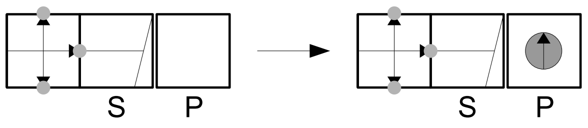
If P has a *fuse* part F as a diagonal neighbour and *F.primary* points toward a neighbour Q of P and the input of F faces an output of a *wire* or *nor* part outputting a *True* signal and $P.location - Q.location = F.secondary$ then at $t+1$ P and Q will be connected.



If P has an *unfuse* part F as a diagonal neighbour and $F.primary$ points toward a neighbour Q of P and the input of F faces an output of a *wire* or *nor* part outputting a *True* signal and $P.location - Q.location = F.secondary$ then at $t+1$ P and Q will be disconnected.



If P has a *slide* part S as a neighbour and $S.primary$ points toward P and the input of S faces an output of a *wire* or *nor* part outputting a *True* signal then mark P as a part to be moved in the direction of $S.secondary$.



The shape of the neighbourhood used by this set of rules is shown in Fig. 2.

The last rule above concerns the action of *slide* parts and introduces the possibility of a part being marked as due to move in a particular direction. The PROPAGATION set of rules, listed below, propagates this marking information.

If P is connected to neighbour Q and Q is marked as due to move in direction d then mark P as due to move in direction d.

If P has a neighbour Q in direction e and Q is marked as due to move in direction $d = \text{opposite}(e)$ then mark P as due to move in direction d.

The MOVEMENT set of rules then uses the marking information to carry out movement:

If a cell C has a neighbour P in direction e and P is marked as due to move in direction $d = \text{opposite}(e)$ then P will be moved into C.

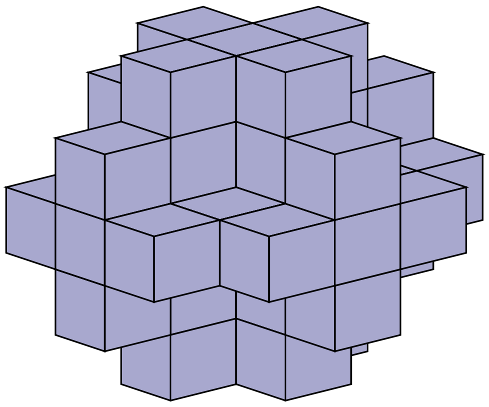


Fig. 2. The shape of the neighbourhood used by the NORMAL set of rules.

Both the PROPAGATION and MOVEMENT sets have a 3D von Neumann neighbourhood.

In order to decide when to switch between one set of CA rules and another we need to know the following information about the environment:

Condition 1 There is at least one part in the environment that is marked as due to move.

Condition 2 Running the PROPAGATION set of rules will not result in any changes (i.e. propagation is complete).

Simulation proceeds using the NORMAL set until Condition 1 becomes true. Then simulation proceeds using the PROPAGATION set until Condition 2 becomes true. Then a single iteration of the MOVEMENT set is run, before switching back to the NORMAL set.

A readable explanation of Gosper's algorithm is given in reference [14]. Extrapolating directly from Gosper's 2D algorithm, a 3D *hashlife* algorithm for a CA with a 3D Moore or von Neumann neighbourhood uses an octree data structure to represent the state of a universe, where a level-n node in the tree represents a cube 2^n cells on a side. Each node contains pointers to eight subcubes, each 2^{n-1} cells on a side. Whenever two or more level-n nodes would represent identical configurations of cells, a single node is used with two or more pointers at level-$n+1$ pointing to the same node. A RESULT pointer either points to nothing, or else to a cube 2^{n-1} cells on a side containing the result of simulating the inner cube 2^{n-2} steps into the future. Whenever a node is formed, a hash value is calculated based on its contents and it is stored in a hash table for future use should an identical node be formed in future. The main advantage conferred by the hash table is that nodes within it are likely to contain already-computed RESULT pointers.

Whereas Gosper's algorithm simulates 2^{n-2} steps forward for a level-n cube, we cannot do this for the CBlocks3D environment. One reason for this is that the NORMAL rule set has a neighbourhood with some cells two units away from the central cell. Therefore calculations of the future state of level-0 nodes may require information from neighbours that are not visible until level-3. For this reason the 4 by 4 by 4 cube that is the result of simulating an 8 by 8 by 8 cube is one time step into the future of the larger cube (rather than 2 as would be the case for a Moore or von Neumann neighbourhood). Furthermore if we are stepping forward in time using the NORMAL rule set at a rate of k time steps per octree calculation step then at some time between t and $t + k$ we may need to switch to the PROPAGATION set. We need to detect when this occurs, back-track to time t and then step forward more slowly until we reach the time when we need to switch to the PROPAGATION set. To reduce the amount of back-tracking that is required, we limit the distance into the future that the algorithm can calculate to 16 time units by ensuring that a doubling of the simulation time-step-size only occurs at levels 4,5,6 and 7 (whereas in *hashlife* doubling occurs at every level). Because of the above mentioned need to back-track and step forward slowly, we also allow results to be calculated with no time-doubling at all, so that simulation can proceed at a rate of one time step per octree calculation step.

Condition 1 and Condition 2 mentioned above can be calculated in a hierarchical way by storing these conditions in each node, and then combining them to calculate the conditions for a higher level node.

These considerations lead to the following structure for a single octree node, expressed using the C programming language:

```
#define EMPTY_FLAG      1
#define NOTPROP_FLAG    2
#define PROPSTATIC_FLAG 4

#define CALCINDEX_CALC  0
#define CALCINDEX_CALC2 1

typedef struct otNode
{
    unsigned char level;
    unsigned char flags;
    otNode *calc[2];

    union
    {
        block leaf;
        struct otNode *children[2][2][2];
    } u;

    struct otNode *next;
} otNode;
```

The *level* member is not strictly necessary, since so long as the level of the highest level node is known, any algorithm operating on a universe can work out which level it is operating on.

The *flags* member has bits for recording whether or not the node is empty (EMPTY_FLAG), whether or not Condition 1 is true (NOTPROP_FLAG) and whether or not Condition 2 is true (PROPSTATIC_FLAG). The set/reset sense of each of these flags is chosen so that flags from sub-nodes can be combined by a bitwise AND operation.

The *calc* member is used to store the result of simulating either one application of a CA rule set (CALCINDEX_CALC) or multiple applications (CALCINDEX_CALC2). Note that it is not necessary to have separate members of *calc* for the NORMAL, PROPAGATION and MOVEMENT rule sets because for a given configuration of cells only one of these rule sets will result in a change. Specifically, the NORMAL rule set will never be applied to any configuration of cells containing cells marked for movement. The PROPAGATION and MOVEMENT rule sets, when applied to any configuration of cells not containing any cells marked for movement, will result in no change. The PROPAGATION rule set will also result in no change when applied to a configuration of cells where movement information is already fully propagated. When the application of a rule set to a configuration is known to result in no change the *calc* member is not used, instead the unchanging result is formed from the central sub-sub-nodes of the node in question.

For level-0 nodes the *leaf* member contains all information about the state of a single cell. This information fits into 18 bits as follows:

Part type: 3 bits
Part orientation: 5 bits
Output state: 1 bit
Connectivity state: 6 bits
Movement propagation direction: 3 bits

For higher level nodes the *children* member contains pointers to subnodes.

The *next* member is not part of the representation of an environment, but is used for making lists of nodes which are used both in the hash table and also for keeping track of unused nodes.

5 Discussion

5.1 Conflicting operations

The CA rule sets described in Sect. 4 do not behave in a consistent way when an attempt is made to carry out two or more conflicting operations on a part at once, such as attempting to move a part in two different directions at the same time. The original implementation of the CBlocks3D environment did cater for these situations, and they were dealt with as follows:

- Any attempt to both connect and disconnect two parts at the same time results in no change in their connectivity state.
- Any attempt to rotate a part about two different axes at the same time results in no change in its orientation.
- Any attempt to move a part in two different directions at the same time results in no movement of the part in question – effectively the movement operations that caused the conflict are cancelled.
- Any attempt to move two or more parts into a single empty cell results in no movement of the parts in question – effectively the movement operations that caused the conflict are cancelled.

The two movement-conflict situations listed above are more subtle than they may at first appear. A comprehensive discussion of the issues involved was carried out by Arbib [15]. Arbib attempted to devise a resolution strategy that is more complex than the one described above and which aims to produce behaviour that more closely approximates the motion of rigid bodies under Newtonian mechanics within the limits of a discrete space environment. For example, Arbib stipulated that an attempt to move a part in two orthogonal directions would succeed and would result in the part moving diagonally. His detailed analysis ends with the sentence 'I hope this formulation is contradiction free', so it is not clear whether he succeeded.

Clearly there is more than one choice of conflict resolution strategy and the extent to which movement-conflicts can be efficiently handled within a given simulation framework depends upon the choice of strategy, which in turn depends upon the reason why a system is being simulated.

One of the simplest strategies is not to resolve movement-conflicts or any other conflicts at all, but instead to declare, as part of the definition of the CBlocks3D universe, that conflicts are illegal and that no well designed mechanism should cause them to arise. This was done in the context in which the CBlocks3D environment was first used [11]. This approach is adequate when using the environment to simulate well designed mechanisms with predictable behaviour, but would not be suitable if one were simulating a system with unpredictable behaviour where conflicts could not be ruled out.

5.2 Performance

The original simulator for the CBlocks3D environment, mentioned in Sect. 3, was capable of simulating the SRPC system at an average rate of about 50 iterations per second on the hardware available at the time. At this rate simulation for a complete replication cycle (220 million iterations) would have taken over six weeks. On the same computer system, using 3.5 Gb of RAM for the hash table, a *hashlife*-based algorithm was able to carry out the same simulation in 10 days at an average rate of 250 iterations per second. The algorithm has been improved since this simulation was carried out, so the same simulation would now be expected to take about 4-5 days.

Table 2 shows the effect that varying the memory available to the algorithm has on performance. This table is based on simulating the first 2 million time steps of the operation of the SRPC mentioned at the end of Sect. 2.

Number of nodes available	Memory used	Iterations per second
2×10^6	88 Mb	125
4×10^6	175 Mb	201
6×10^6	263 Mb	272
8×10^6	350 Mb	313
10×10^6	439 Mb	345
15×10^6	658 Mb	396
20×10^6	877 Mb	428
30×10^6	1316 Mb	559
40×10^6	1754 Mb	563
50×10^6	2193 Mb	620

Table 2. Effect of varying the number of nodes on simulation speed.

The complete source code for the algorithm described in this paper is available from:

`http://www.srm.org.uk/downloads/CBlocks3DHash.zip`

6 Acknowledgements

Thanks to the anonymous reviewers of this paper for valuable suggestions. Most of the work in this paper was undertaken as part of an external PhD research project with the Department of Physics and Astronomy, Open University, UK.

References

1. Gosper, R.W.: Exploiting regularities in large cellular spaces. Physica D. **10(1-2)** 75–80 (1984)
2. Golly: a cross-platform application for exploring John Conway's Game of Life and other cellular automata. http://golly.sourceforge.net (visited on 14th June 2010)
3. Pesavento, U.: An Implementation of von Neumann's Self-Reproducing Machine. Artificial Life **2(4)** 337–354 (1995)
4. Nobili, R.: The Cellular Automata of John von Neumann. http://www.pd.infn.it/~rnobili/wjvn/index.htm (visited on 14th June 2010)
5. Buckley, W.R.: Signal Crossing Solutions in von Neumann self-replicating cellular automata. Automata 2008 453–501, Luniver Press, Frome, UK (2008)
6. Hutton, T.J.: Codd's Self-Replicating Computer. Artificial Life **16(2)** 99–117 (2010)
7. Stevens, W.M.: Simulating Self Replicating Machines. J. Intelligent and Robotic Sys. **49(2)** 135–150 (2007)

8. Burks, A.W., von Neumann, J.: Theory of Self-Reproducing Automata (pages 81–82). University of Illinois Press, Urbana, Illinois (1966)
9. Johnson, D. L.: Elements of logic via numbers and sets (pages 53–65). Springer-Verlag, London (1998)
10. Online Wikibooks chapter on sets: http://en.wikibooks.org/wiki/Set_Theory/Sets (visited on 14th June 2010).
11. Stevens, W.M.: Self-replication, construction and computation. PhD Thesis, Open University, Milton Keynes, UK (2009)
12. Stevens, W.M.: A Self-Replicating Programmable Constructor in a Kinematic Simulation Environment. Robotica, Cambridge University Press (in press)
13. Stevens, W.M.: Parts closure in a kinematic self-replicating programmable constructor. Artificial Life and Robotics. **13(2)** 508–511 (2009)
14. Rokicki, T.G.: An algorithm for compressing space and time. Dr Dobb's Journal **31(4)** 12–18 (2006)
15. Arbib, M.A.: Machines which Compute and Construct. In: Theories of Abstract Automata 355–361, Prentice-Hall, Englewood Cliffs, New Jersey (1969)