Remote Execution and 3D Visualization of Cell-DEVS models

Gabriel Wainer Wenhong Chen

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building. 1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.
gwainer@sce.carleton.ca

Keywords: Cell-DEVS models, web-based simulation, cellular automata, Virtual Reality, VRML.

Abstract

The CD++ tool was created to simulate complex physical systems using a cell-based approach. The original visualization facilities of this tool were too limited. We extended them using VRML in order to provide a 3D graphical interface to allow the users to analyze execution results with ease. The tool was built using a client/server architecture, therefore, users can easily input a simulation, execute it in a remote CD++ server, then receive, visualize and analyze the results locally with easy-to-use web-based 2D and 3D interfaces. The client can also support multi-view simulation, and run several different models simultaneously.

1. INTRODUCTION

Simulation is becoming increasingly important in the analysis and design of complex systems. Scientists and engineers have long used models to better understand the systems they are studying, for analysis, understanding, prediction and design of different complex physical phenomena. At present, there is a large number of modeling and simulation techniques, and various types of simulation tools have been developed to deal with complex systems and the interactions among their constituent parts. A formalism that is gaining popularity in recent years is called **Discrete Event Systems Specification (DEVS)** [1], a framework for the construction of discrete-event hierarchical modular models, allowing for model reusing. In DEVS, basic models (**atomic**) are specified as black boxes, and they can be integrated together forming a hierarchical structural model (**coupled**).

Cell-DEVS [2] extended the DEVS formalism allowing to simulate discrete-event cellular models. The approach extends traditional Cellular Automata (CA) [3], which are defined as a lattice of cells updated synchronously and simultaneously. Each cell in a CA has a state value and a local rule that defines how to obtain a new value based on the current state and the values of neighboring cells. Cell-DEVS extends these concepts by defining a cell as a DEVS atomic model and a cell space as a DEVS coupled model, and by introducing a new way of defining the timing of each cell that is more flexible than the existing approaches.

The CD++ tool [4] enables simulating DEVS and Cell-DEVS, and it has been used to create a variety of models in different areas: biology (watersheds, fire spread, ant colonies), physics (crystal growth, lattice gases, heat diffusion), chemistry (solution diffusion in moving fluids), and several artificial systems (autonomous robots, heat seekers, urban traffic, etc.) [5, 6]. While running these models, we found out that, when the complexity of the models increase, the execution requires a computing power that standard personal computers do not provide. Despite this fact, end users might not have access to high performance computing resources, and they might need to use workstations for analysis and development.

A solution to these problems is to let the user to execute simulation software on a high-performance remote computer, while using a workstation for development and analysis. In any of these cases, a client/server architecture is adequate. The computer running the simulation software can be designed as a server and execute many simulation models simultaneously to serve many users at the same time. CD++ design was modified following this idea, and it was transformed into a simulation server that receives requests from many users and serves them simultaneously. The separation of concerns provided by the DEVS and Cell-DEVS formalisms permitted us to attack these changes with ease. In DEVS-based modeling environments, models are completely independent from the simulation engines, and simulators can be exchanged without doing any modifications to existing models.

Another problem of CD++ was the lack of adequate visualization mechanisms. Visualization tools are crucial in helping to better understand the behavior of complex systems, facilitating thinking, problem solving, and decision making, scientific visualization tools create visual displays, in which numeric values in data sets are represented visually as colors, shapes, or symbols [7].

We introduce an extension for CD++ that enables 2D and 3D visualization, and remote access to a DEVS simulation server. The end user tools are organized as a simulation client applied to the CD++ toolkit. A 3D visualization GUI is developed using VRML and Java, providing useful functions for the users. A multi-view GUI permits displaying multiple views of the result (any number of areas of interest of the same result can be displayed at the same time). Using these facilities, the users can now develop and test their models in local workstations, and send them to be simulated in a remote CD++ server executing in a high performance platform. Then, they can receive, visualize and analyze the result on the local computer, improving model definition and execution.

2. DEVS AND CELL-DEVS

The DEVS formalism [1] was originally defined in the '70s as a discrete-event modeling specification mechanism. A real system modeled with DEVS is defined of a composite of sub-models, each of them being atomic or coupled. Each model is defined by a time base, states, inputs, outputs, and the functions to determine the next states and outputs. Coupled models can be integrated into a model hierarchy, allowing model reuse, reducing testing time and improving productivity.

A DEVS atomic model is described as:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

The model is seen as having an interface consisting of input (*X*) and output (*Y*) ports to interact with other models. External input events arrive through input ports, and trigger the external transition function d_{ext}. The internal transition d_{int} is activated after the model spends a specific duration at the present state **S**, which is defined by the duration function **D**. The transition will produce internal state changes after the results are sent out through output ports. These results are determined by the output function l.

A DEVS coupled model is composed of several atomic or coupled sub-models, and can be defined as:

$$CM = < X, Y, D, \{Mi\}, \{Ii\}, \{Zij\} >$$

The coupled models consist of a set of basic components **D** (atomic or coupled model), which interact with each other through the model's interfaces(***X, Y***). The model **Mi** sends the outputs to its influencees **Ii**. The translation function **Zij** ($\forall j \in Ii$) converts the outputs of a model into inputs of the other models.

The Cell-DEVS formalism extended this basic behavior to allow the implementation of cellular models with timing delays [2]. Each cell in these spaces holds a state variable, which is updated according to a local rule that considers the present cell state and those of a finite set of nearby cells (called the cell's **neighborhood**). Each cell is defined as an atomic model with timing delays, and it can be integrated to a coupled model to represent a cell space.

Cell-DEVS defines cells as atomic model, specified as:

$$TDC = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

Where *X* is a set of external input events, *Y* a set of external output events, *I* the model's modular interface, S the set of sequential states for the cell, θ the cell state definition, N the set of input values, d the delay for the cell, δ_{int} the internal transition function, δ_{ext} the external transition function, τ the local computation function, λ the output function; and D the state's duration function. A cell uses the input values N to compute its next state, which is obtained by applying the local computation function τ. A delay function is associated with each cell, deferring the computed result to be sent to the neighbor cells. There are two types of delays: inertial and transport delay. For the transport delay, the next value is added to a queue sorted by output time; therefore, the results can be stored until they have been sent out. On the contrary, inertial delay uses a preemptive policy, that is, any stored previous outputs will be deleted, if there are changes before the delay consumption. The δ_{ext} function activates the local computation.

After the basic behavior of a cell is defined, the whole cell space can be constructed by building a coupled Cell-DEVS model, defined by:

$$GCC = < Xlist, Ylist, I, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z >$$

Where **Xlist** is the input coupling list, **Ylist** the output coupling list, **I** the definition of the interface for the modular model, ***X*** the set of external input events, ***Y*** the set of external output events, **n** the dimension of the cell space, $\mathbf{\{t_1,...,t_n\}}$ the number of cells in each of the dimensions, **N** the neighborhood set, **C** the cell space, **B** the set of border cells; and **Z** the translation function. This specification defines a coupled model composed of an array of atomic cells. Each cell is connected to the cells in its neighborhood. Since a cell space is finite, the cells on the borders should have a different neighborhood than the rest of the space, they are "wrapped", that is, cells on the border are connected to the cells in the opposite one. Finally, the Z function defines the internal and external coupling of cells in the model. This function translates the outputs of m-eth output port in cell C_{ij} into values for the m-eth input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood.

The **CD++** environment [4] implements the DEVS and Cell-DEVS theory. The toolkit has been built as a set of independent software pieces, each of them independent of the operating environment chosen. There are versions running under Windows 95/NT, Linux, AIX, IRIX, HP-UX and Solaris. It allows defining models according to the specifications introduced in the previous section. A set of independent applications related with the tool allows the users to have a complete toolkit to be applied in the development of simulation models. CD++ can be used to define Cell-DEVS models. The tool includes an interpreter for a specification language that allows describing the behavior of each cell of a cellular model, including its delay and neighborhood. In addition, it allows defining the size of the cell space and their connection with other DEVS models, the border and the initial state of each cell. The behavior specification for a cell is defined using a set of rules, each indicating the value for the cell's state if a condition is satisfied. The output of the model should be delayed by using a specified time.

For instance, Figure 1 shows an example for the specification of a Cell-DEVS model developed using CD++. The specification follows Cell-DEVS formal definitions. In this case, Xlist = Ylist = {∅}. The set {t1…tn} is here defined by *width-height*, which specify the size of the cell space (in this example, n=2, t1=20, t2=40). The N set is defined by the lines using the *neighborhood* keyword. The border (B) is wrapped. Using this information, the tool builds a cell space (specified by C in the formal specification), and the Z translation function. Each cell in the cell space is built following the Cell-DEVS specifications. The X, Y, S, N, θ, δ_{int}, δ_{ext}, λ, and D functions are built following Cell-DEVS formal specifications (see [2] for details). The user only needs to define

the τ function (defined by *localtransition*), and the delay (defined by *delay* and *defaultDelayTime*). In this example, the local computing function executes very simple rules. The first one indicates that, whenever a cell state is 1 and the sum of the state values in N is 8 or 10, the cell state remain in 1. This state change will be spread to the neighboring cells after 100 ms. The second rule states that, whenever a cell state is 0 and the sum of the inputs is larger or equal to 10, the cell value changes to 1. In any other case, the result remains unchanged (*t* = true), and it will be spread to the neighbors after 150 ms. As we can see, cells evolve using a discrete-event approach.

```
[ex]
type : cell
width : 20     height : 40
delay : transport     border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1)
neighbors : (0,-1)  (0,0)  (0,1)
neighbors : (1,-1)  (1,0)  (1,1)
localtransition : rules

[rules]
rule : 1 100 { (0,0) = 1 and (truecount = 8 or
truecount = 10) }
rule : 1 200 { (0,0) = 0 and truecount >= 10 }
rule : (0,0) 150 { t }
```

Figure 1. A Cell-DEVS specification in CD++

CD++ generates a text file representing a 2-dimensional slice showing the execution of Cell-DEVS models as a matrix of values. In models of three or more dimensions, the results can be shown as slices representing 2-dimensional planes, each of them shown as a matrix. For instance, in a 3D dimensional space, the first plane shown corresponds to (x, y, 0), the second one to (x, y, 1), etc. Figure 2 shows a model executing a 3D simulation of the 'Life' game [8] with the original rules proposed by Conway. Each cell can be alive (1) or dead (0). A new cell is born when it has exactly three living neighbors. An existing cell survives if it has two or three neighbors that are alive. Otherwise, it dies.

```
Line : 247 - Time: 00:00:00:000
     0123456          0123456          0123456
    +-------+        +-------+        +-------+
   0|1      |       0|       |       0|1      |
   1|1 1  11|       1|11   11|       1|  111  |
   2| 1   1 |       2|   11 1|       2| 1 11  |
   3|       |       3|  1  11|       3|     11|
   4|  1  11|       4|  1 1  |       4| 1   11|
   5|  11  1|       5|   1 1 |       4| 11  1 |
   6|1  1  1|       6| 1   1 |       4| 1 11 1|
    +-------+        +-------+        +-------+

Line : 247 - Time: 00:00:00:100
     0123456          0123456          0123456

    +-------+        +-------+        +-------+
   0| 1    1|       0|11    1|       0| 1    1|
   1|1 1   1|       1|1     1|       1|1 11  1|
   2|11  1 1|       2|1    1 |       2|11   11|
   3|    111|       3|  1 1 1|       3|    1 1|
   4|       |       4|     11|       4|       |
   5|1  111 |       5|1 111 1|       4|1  11 1|
   6|       |       6| 1     |       4| 1  1 1|
    +-------+        +-------+        +-------+
```

Figure 2. A fragment of a 3D model

As mentioned in section 1, we are interested in running a simulation using a client/server architecture. Therefore, VRML (Virtual Reality Modeling Language) [9] appears to be a good choice to our goals. VRML is a web-based graphics language for creating 3D models. It is a file format for describing 3D objects and worlds, and allows user interaction within a scene through viewpoints, movement, and rotation. VRML worlds are created with a scene-graph structure. Scene graphs are comprised of various groups of nodes, which are responsible for displaying shapes, interaction, and navigation through the world. The External Authoring Interface (EAI) [10] is a Java Application Programming Interface (API) that enables a currently running applet to interact with, and dynamically update a 3D VRML scene. Using Java and the EAI, the users can have full control to create a dynamic 3D VRML World.

3. CD++ SIMULATION CLIENT/SERVER

Considering the background considered in the previous section, we decided to provide a set of tools for building Cell-DEVS simulation models, visualizing and analyzing the results, and accessing a remote simulation server for execution. The idea was to incorporate all of these components together as a simulation client in a client/server architecture. The inherent modularity of client/server systems leads to greater overall robustness, since computing responsibility is no longer concentrated in the server. Client/Server applications typically distribute the software components so that the data source resides on the server, the user interface resides on the client, and the logic resides in either side.

Following these ideas, the CD++ simulator was modified to run as a stand-alone application or as a server. When CD++ runs as a server, the components are related as in figure 3. There is a separation between model definition, simulation execution, and visualization. A user run CD++ on a local machine and specifies a model file as the input. After the simulation is over, a log file like the one in figure 2 is generated. These files can be used to generate graphical outputs using different Graphical User Interfaces (GUI).

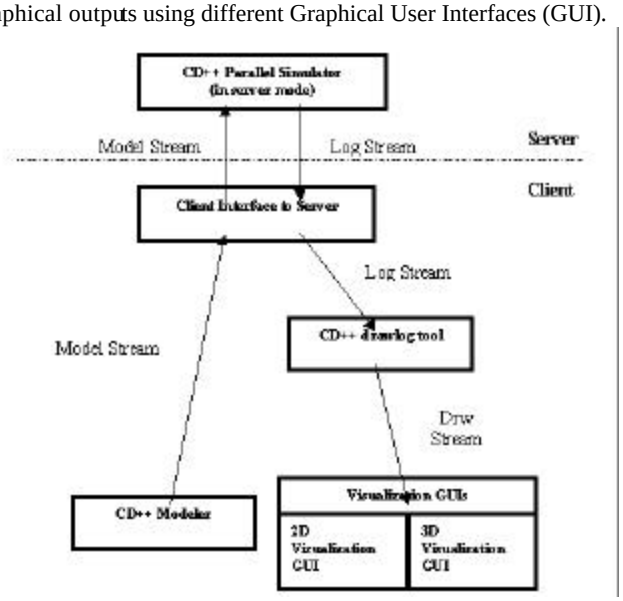


Figure 3. CD++ running as client/server application.

The client integrates components for model input and result visualization GUIs, which are the main interfaces on the client. They ensure that the user is presented with a visualization system that has a familiar, easy-to-use interface, requiring little training overhead. In this way, CD++ simulations can be presented to a wider range of users, and learning times and training costs can be significantly reduced

When CD++ is running as a server, it is always listening on a specified TCP socket. The clients needing service will communicate with it and will send model files through this port. The server allows to service several clients at the same time, and whenever it accepts a simulation requirement, a child process is created to serve the specific requirement. A child process created most recently takes the place of its parent and continues listening through the desired port. When a new incoming simulation arrives, the described process is repeated; the CD++ server is forked and the simulation request is satisfied. This process is terminated when the simulation is over. Concurrent simulation requests can be serviced. In order to execute a simulation, the server must send a *model* file like the one defined in Figure 1, an optional *event list* and an optional *stop time*. When a request is received, CD++ executes the model, and it returns the result through the same TCP port. with the following format

The CD++ server will send back the log file, and the client will receive and save it on the local disk. CD++ drawlog can be used to change the format of a log file to a text file that can be used for visualization.

4. 3D VISUALIZATION ENVIRONMENT

The 3D visualization GUI is used to visualize the result in a 3D environment. The results are represented using nodes in the VRML scene. The user can navigate in the scene, and edit the nodes in the scene for more convenient investigation on the result.

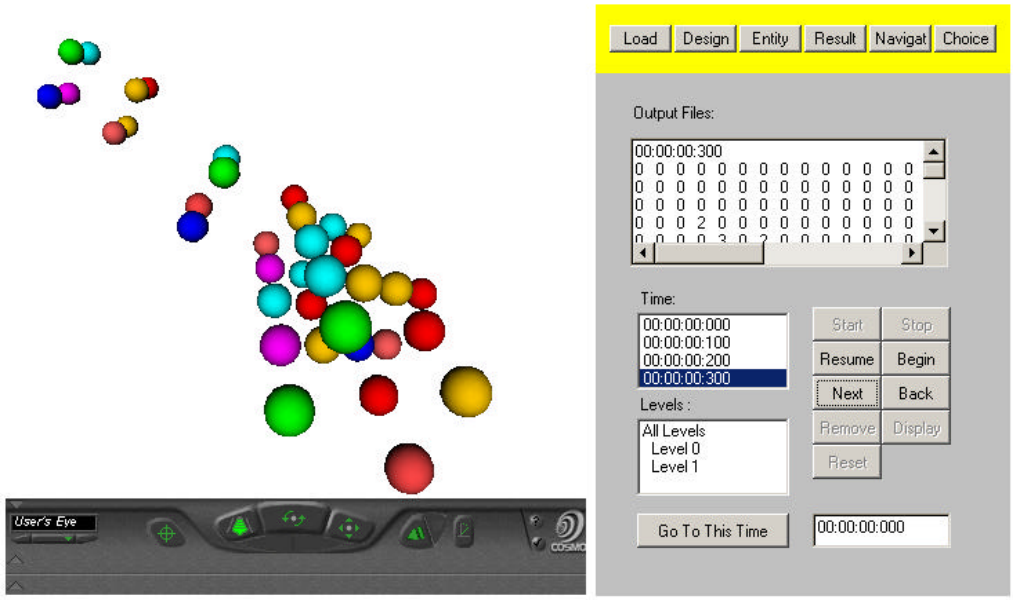


Figure 4. NavigatePanel execution.

We start with an empty VRML scene, holding nodes that can be added and removed from it to represent the results. This empty VRML scene is embedded in an HTML file as a root file, which also includes an applet to control the scene. The applet includes functions to add or remove nodes, update the scene, navigate, and edit the nodes in the scene. The applet has the functions:

1. Add or remove a node from the scene
2. Change the shape and colors of the nodes, or hide the nodes
3. Select the colors for the value ranges of the nodes
4. Change the scale of the nodes, and the interval between them
5. Navigate in the Scene
6. Edit the scene and the individual node
7. Load a result file to be displayed
8.

These functions are classified into several categories:

- **ReadDrwFile** class: it is in charge of reading the next recent result in the result file. It is used by the NavigatePanel to get the next result to be displayed. This result can be chosen to be: a) The result with the next timestamp in the result file; b) The result with the previous timestamp; c) The result at a chosen time.

- **InfoPanel**: it is a subclass of Panel, which is shown when the application starts. It includes methods to select the result file, and an associated color palette. These files are checked for consistency, and a text area displays debugging and status information.

- **NavigatPanel**: it is the main class in the application. It stores the currently displayed result, recent displayed nodes, and the names of all the displayed nodes. This information is updated whenever the scene is updated with a new result, or the color palette is changed. It first initiates the scene, using the number of rows,

columns and layers. Then, the scene is initiated as a block of nodes corresponding to the size of rows, columns and layers. Each node in the block represents a value in the result. The class includes methods to add or remove a node in the scene, and to change the shape, color, and size of the nodes, or to check the result at a favorite viewpoint. In order to change the shape, color, and size of the nodes, we just need to call the functions in the node class. Another function provided by this class is the *navigation*, which is implemented as the possibility of having different viewpoints. First, the viewpoint can be a child of Transform node, its position and orientation are changed with those in this Transform node. Second, the user can select different viewpoints. If another viewpoint is bound, the active viewpoint will be unbound automatically, and the user will see the scene corresponding to the newly bound viewpoint. This class has several methods for the navigation in the visualization. We just need to get the position of the next recent result, and call the method in the ReadDrwFile class. We can also read the next result, advance to the next result, move to the previous result, move to any selected time, or input time, reset the visualization again and pick a one node as the current node. Figure 4 shows the output generated by the NavigatPanel class.

- **EntityPanel**: it allows editing individual nodes in the scene. The entity list is populated by the NavigatPanel class when it updates the nodes in the scene every time, and a node can become the current node by clicking it or selecting the corresponding item in a list. After a node becomes the current node, it can be edited by calling the related methods in the node class and the methods in NavigatPanel class. The functions included in this class are:

1. Change the shape, color, and size of the selected node
2. Add or remove the individual node in the scene.
3. Set the slides according to the values of the selected node.

- **ResultPanel**: it is used to navigate in the VRML world. Different methods are defined to control the navigation, including: a) A start method; b) a resume method to continue if stopped; c) a go back method to go to the previous timestamp; d) a go next method to go to the next time; e) a stop method; f) a continuously display method (this iteration will end only at the end of file); g) a method to go to any selected timestamp; a) a method to remove layers in the scene or add the removed layers; i) a reset method for the scene, displaying all the layers. These methods call the corresponding methods in the ResultPanel Class, which will call the corresponding methods in the NavigatPanel Class.

5. VISUALIZING CD++ SIMULATIONS

In this section, we show some examples of execution of models using the VRML 3D interface. We will not focus in the model execution results (see [4, 5] for details), but in the client functionality.

The result matrix is now shown as a 3D matrix of colored nodes with the same size, and the layer displayed as an another node matrix with different value in z coordinate. Each node corresponds to a value in the result matrix, and the color of the node is specified by its value and set with a color selection.

We can use different geometries to represent the objects in the result. The user can select boxes, spheres, cones or cylinders. Two examples can be seen in Figure 5. The result matrix is displayed as a node matrix, and the layer displayed as an another node matrix with different value in z coordinate. We can also see that different colors are assigned to the nodes using a palette selection.

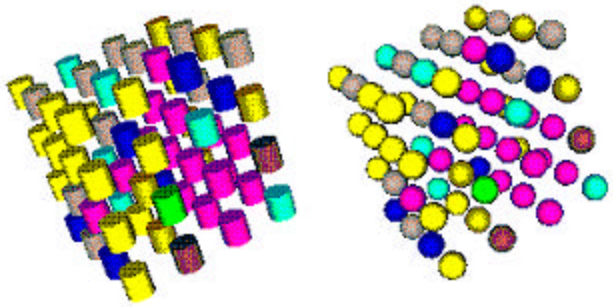


Figure 5. Changing geometry

Another facility enables selecting different viewpoints to visualize the results. This can be seen in figure 6.

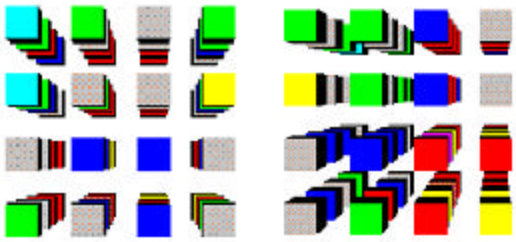


Figure 6. Viewpoints

A user can select any viewpoint defined in the VRML file to visualize the result. Here, we show the *User's Eye* and the *Side view* viewpoints. In addition, the user can select any viewing area, as shown in previous figures. The navigation facilities enable displaying the result continuously with a sequence same as the simulation. In this way, the user can see the results continuously, with the same sequence than the simulation. The user can also edit a single node in the scene, changing its shape, color or position, as shown in the following figure.

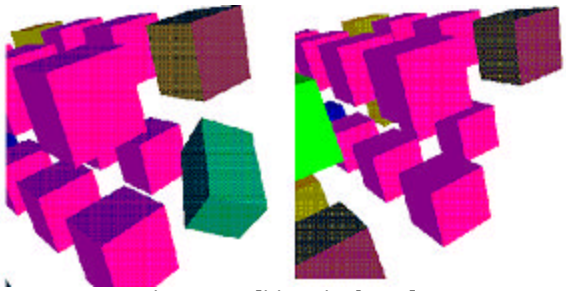
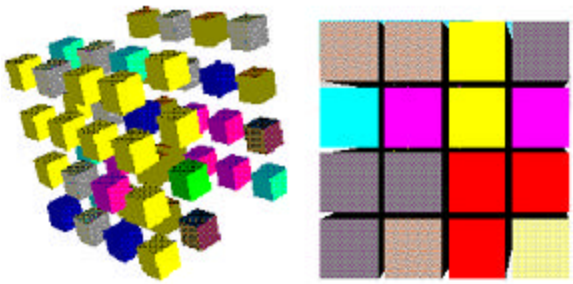


Figure 7. Editing single nodes

The edited node will keep the modified attributes, such as, color and size. Therefore, the user can highlight the special nodes he

wants to check. (1) Modify a node with color, size and translation and rotation. (2) Delete the edited node. The user also can redisplay the deleted node, then the display will be as in (1), and all the modified attributes are remained.

A user can also remove any layer in the display, in order to make easier the visualization of certain phenomena. In figure 8, we show the previous examples, but level 1 was removed, which can be redisplayed later if needed.



(a) **(b)**

Figure 8. (a) Deleting layers (b) scaling nodes

The nodes in the scene can be scaled up or down, as shown in Figure 8.b), where the nodes have been scaled to the minimum distance and cannot be scaled further. The nodes also can be scaled to smaller size.

Finally, multiple instances of the GUI can be activated to visualize the same result, using different viewing areas, as shown in the following figure. Likewise, different geometry or Inline nodes can be used, if needed.

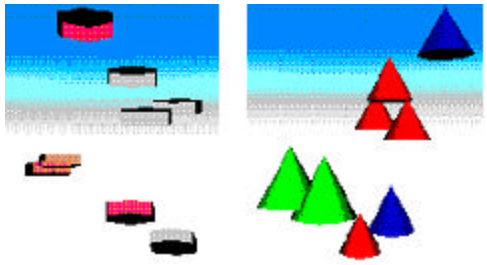


Figure 9. Multiview: different shapes from different viewpoints.

6. CONCLUSION

CD++ is a tool for the simulation of complex physical systems that can be used to simulate a variety of models. To facilitate the users to use the CD++ simulator, this client provides a series of tools, including 3D simulation visualization and remote execution.

The 3D visualization GUI enables sophisticated visualization of Cell-DEVS models. To better understand the results, the user can select shapes to represent a node in the 3D visualization, select different colors or hide for the nodes with different values, edit individual node and remove layers. The interface in this client can send the simulation model to a remote CD++ server, then receive, and visualize the results locally. With this client, a distributed simulation system can be easily established in various environments. This client also can support real-time multi-view, multi-user simulation, and run several different models simultaneously, improving the use of the previously existing tools.

REFERENCES

[1] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.

[2] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modeling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", Springer-Verlag. 2001.

[3] TOFFOLI, T. "Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular automata)". *Proceedings of ACRI'94*. 1994.

[4] WAINER, G. "CD++: a toolkit to define discrete-event models". G. Wainer. *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[5] AMEGHINO, J.; TROCCOLI, A.; WAINER, G. "Modeling and simulation of complex physical systems using Cell-DEVS". In *Proceedings of 34th IEEE/SCS Annual Simulation Symposium.* Seattle, U.S.A. 2001.

[6] AMEGHINO, J.; WAINER, G. "Application of the Cell-DEVS paradigm using N-CD++". J. Ameghino, G. Wainer. In *Proceedings of the 32nd SCS Summer Computer Simulation Conference*. Vancouver, Canada. 2000.

[7] KRAAK, M.-J.; MACEACHREN, A.M. Visualization for exploration of spatial data (editorial introduction to special issue). International Journal of Geographical Information Science, 13(4), 285-287. 1999.

[8] GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'.". *Scientific American*. 23 (4). pp. 120-123. April 1970.

[9] AMES, A.; NADEAU, D.; MORELAND, J. "VRML 2.0 Source (Second Edition)" John Wiley & Sons, Inc 1997

[10] MARRIN, C. "External Authoring Interface Reference", Silicon Graphics, Inc., 1997.
`http://www.web3d.org/WorkingGroups/vrml-eai/history/eai_draft.html`