**3D Visualization of a CD++ Cell-DEVS**

**Minefield Mapping Simulation Using Blender**

**Tania L. Pendergast**

**Department of Systems and Computer Engineering**

**Carleton University**

**1125 Colonel By Drive**

**Ottawa, ON K1S 5B6**

[tpenderg@connect.carleton.ca](mailto:tpenderg@connect.carleton.ca)

**Keywords:** Cell-DEVS, CD++ Toolkit, visualization, Blender, minefield mapping

**Abstract**

Cell-DEVS is an extension of the DEVS formalism which combines DEVS with the formalism for Cellular Automata. It is particularly useful for defining spaces by decomposing them into individual cells. The CD++ Toolkit enables one to model and simulate a real or artificial system using either DEVS or Cell-DEVS. It contains a Modeler tool that permits only 2D visualization of the simulation. This particular paper focuses upon an effort to produce a 3D visualization of an existing CD++ Cell-DEVS simulation using the open source 3D visualization software called Blender. The simulation scenario used was one that involved the mapping of a 10x10 minefield using four robots. The individual robots were constrained to move about the minefield in order to scan the ground to detect the presence or absence of mines. It was assumed that the robots were in contact and were updating a common map of the minefield. This paper discusses the conceptual design and implementation processes, problems encountered along with their resolution, and also provides recommendations for areas of future work.

1. **INTRODUCTION**

The ultimate goal of this work was to take the output from an existing CD++ Cell-DEVS simulation and use it to produce a 3D visualization. The approach that was chosen involved the use of the “[freely available] and open source 3D content creation suite [called Blender]” [1].

This paper will discuss in detail the steps taken to achieve this goal, from initial familiarization with previous work and the tools required, through conceptual design, and finally to implementation. Problems encountered and areas for future work will also be discussed.

1. **BACKGROUND**
   1. **Discrete Event systems Specification formalism**

The Discrete Event systems Specification (DEVS) formalism provides a solid mathematical modeling foundation and it is based upon Systems Theory concepts. When applying the DEVS formalism to the modeling of a real or artificial system, one can decompose it into atomic and coupled components, or models. An atomic model may be considered to be the most basic of building blocks with which to represent a system. Specific states are defined for each atomic model, and the model can exist in only one state at a given point in time. A coupled model is composed of several atomic or coupled models. The DEVS formalism is used to define each model and their hierarchical interconnections. [2]

* 1. **Cell-DEVS**

Cell-DEVS [3] is an extension of the DEVS formalism which combines DEVS with the formalism for Cellular Automata (CA). Using this approach to modeling allows a practitioner to define a cellular space that is made up of individual cells. Each cell is defined as an atomic model and the method for the coupling of these cells is also defined. [4]

A cell’s behaviour is defined using the inputs (N) from its neighbours (which are defined by a particular type of neighbourhood). It is these inputs that activate a cell’s local computing function (τ). The outputs from a cell are controlled by the type of delay (d). Figure 1 provides a depiction of this.



***Figure 1. A Cell-DEVS atomic model with transport delay***

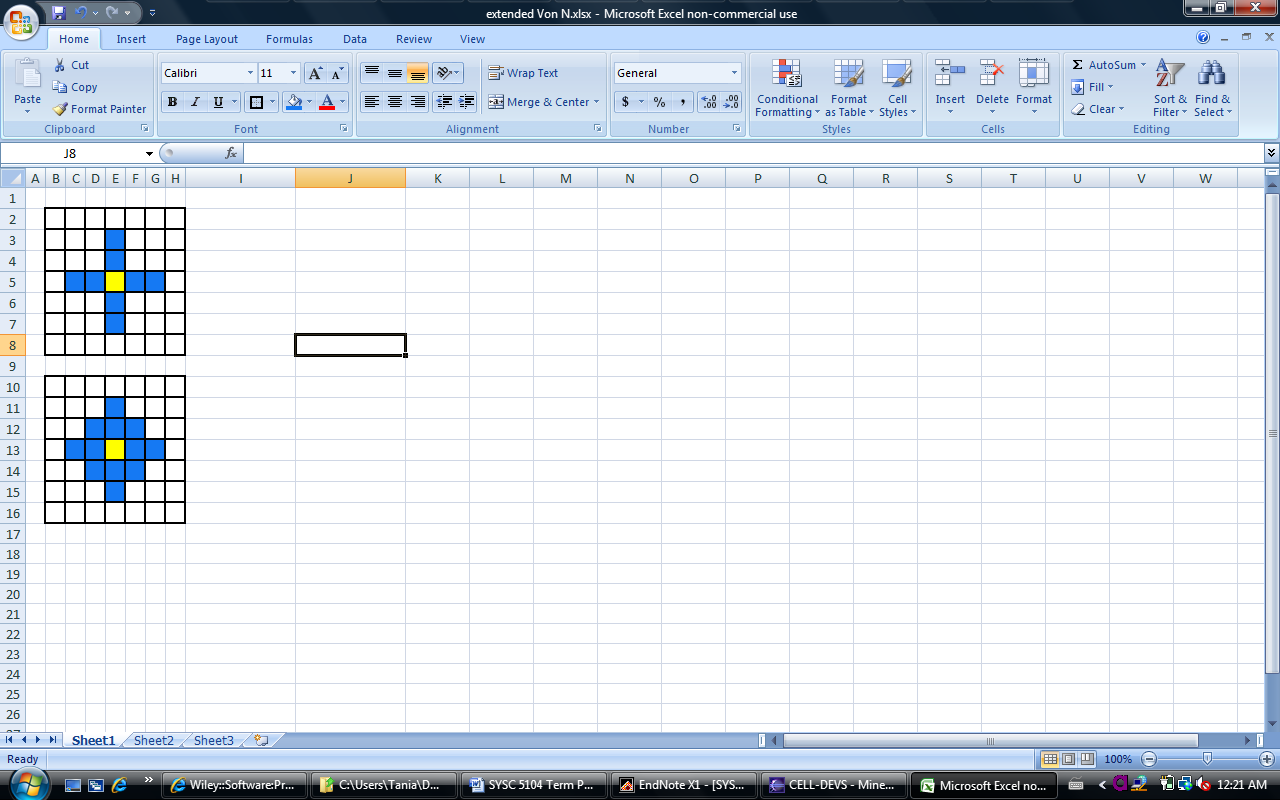
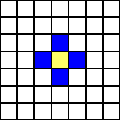
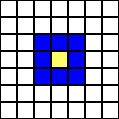
A coupled Cell-DEVS model is the resulting array of cells (atomic models) with a given dimensions, border, and zones (if applicable).



***Figure 2. A Cell-DEVS coupled model***

The key parameters that are used to define a Cell-DEVS model are discussed below:

* **Neighbourhood**: A cell’s neighbourhood is made up of the neighbours whose inputs the cell will receive. The most widely used neighbourhoods from CA include: Moore (includes the 8 adjacent cells and the origin cell); Von Neumann (includes cells to the North, South, East and West and the origin cell); extended Von Neumann (a 5x5 rhombus with origin in the central cell); hexagonal topology; and triangular topology. [4]  
  Figure 3 depicts three of these neighbourhoods.



***Figure 3. Moore, Von Neumann and   
Extended Von Neumann neighbourhoods***

* **Local computing function (τ)**: The function used to compute the future state of the cell based upon the inputs from the neighbourhood.
* **Delay (d)**: A delay can be associated with each cell. It is this delay that defines the time at which state changes are transmitted to neighbouring cells. The delay can be either inertial or transport. Transport delay defines state changes that occur upon the expiration of the delay. Inertial delay introduces the notion of pre-emption. “The last arrived future event can be pre-empted if there is a new input before the consumption of the inertial delay” [4].
* **Dimensions**: The cell space may be two dimensional or include additional layers that provide a greater level of detail to the model.
* **Border**: A cell space can have a defined border or no border. When there is no border, this defines a model where all cells have the same behaviour (ie. the cells are said to be wrapped, or described another way: “cells in one border are connected with those in the opposite one using the neighbourhood relationship” [4]). If a border is defined then the cell space is not wrapped and the associated unique zones must also be defined.
* **Zones**: These are used to define areas of the cell space with unique behaviour. For example, if the Cell-DEVS model represents a room, the border would not be wrapped and zones would be specified to define the unique behaviour of walls, doors, windows, obstacles, etc.
  1. **CD++ Toolkit**

The CD++ Toolkit [5] is a modeling and simulation

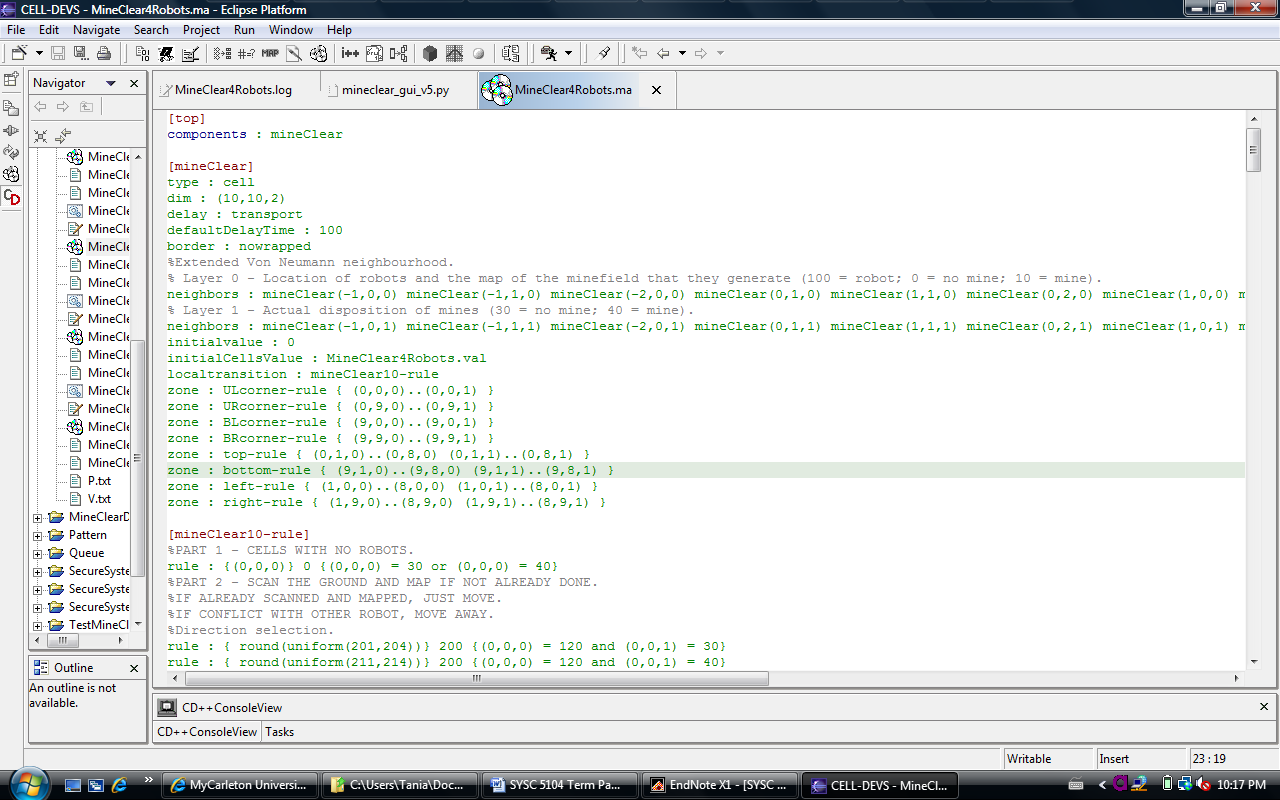
tool that allows for the development and simulation of models in both DEVS and Cell-DEVS. In the case of DEVS, atomic models are programmed in C++ and are incorporated into a basic class hierarchy [6]. Models can then be coupled using a specification language that is built into the simulation engine.

When using a Cell-DEVS approach, the toolkit also includes a specification language for defining models based upon the formal specification. Cell-DEVS models are considered to be “a special case of coupled models [6].” In order to properly characterize the complete cell space, the parameters discussed in section 2.1 must be defined [6]. The behaviour of the local computing function (τ) is captured in a set of rules that follow a given format:

POSTCONDITION DELAY (PRECONDITION)

This means that when the precondition is satisfied, the state of the cell will change to the defined postcondition following the specified delay time. The proper ordering of the rules is significant. If the precondition is found to be *False*, then the next rule in the list will be evaluated. If no rule in the list is found to be *True*, an error will be generated in order to identify that the specification is not complete. [6]

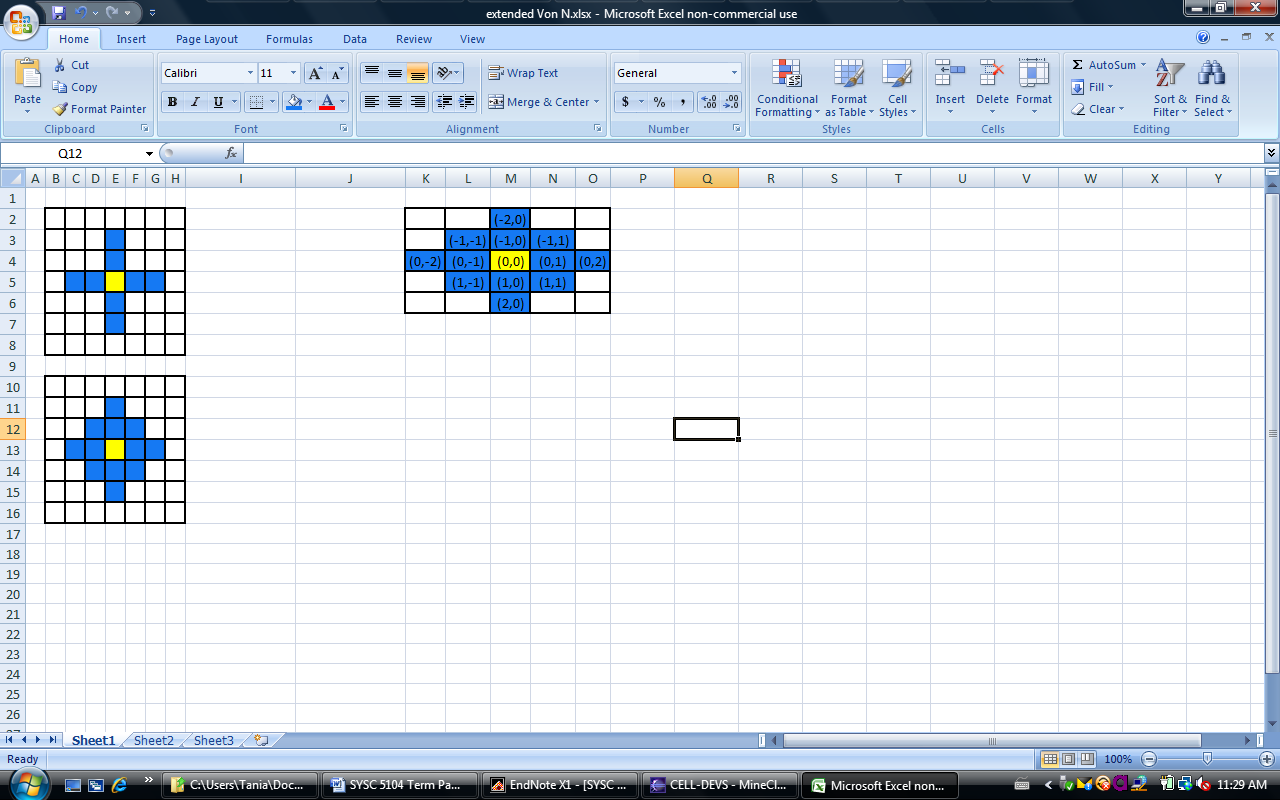
Figure 4 provides an example of the specification language used to define models using the CD++ Toolkit for Cell-DEVS. This particular example was taken from the minefield mapping simulation [7] that will be described in detail in the next section.



***Figure 4. Snapshot of a sample CD++ specification***

Of particular note from Figure 4 are the following defined parameters:

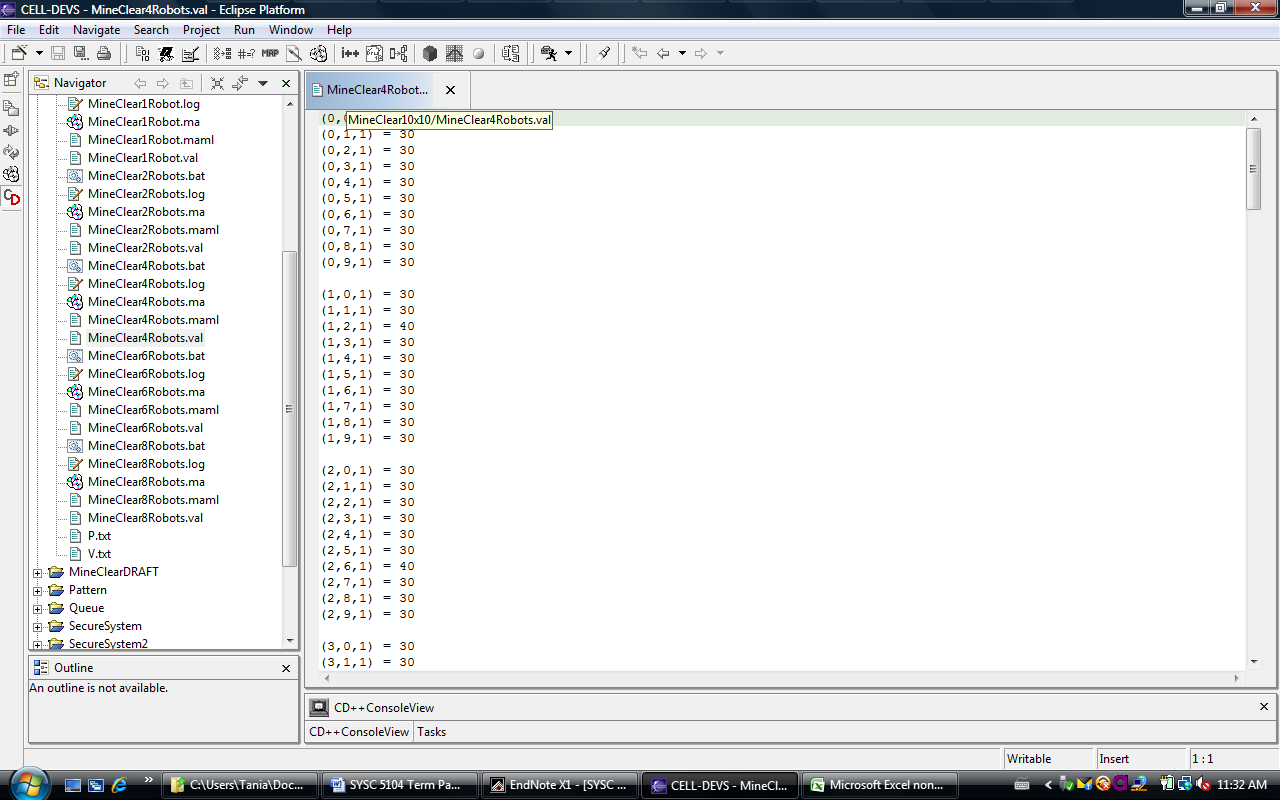
* **Neighbourhood**: defined by specifying each of the neighbours in relation to the cell of origin. “Each cell (*x1,i,...,xn,i*) represents a displacement from the center cell (0,...,0)”[6], where *-x* denotes both up and left, and *+x* denotes down and right, as shown in Figure 5.



***Figure 5. Sample definition of neighbours***

* **Local computing function**: specified through the definition of the rules.
* **Delay**: transport.
* **Dimensions**: 10x10 with two layers.
* **Border**: is not wrapped, so unique zones have been specified.
* **Zones**: have been used to specify the unique behaviour for the four borders of the minefield and each of its four corners. Each zone represents a cell, or group of cells, with its own set of rules.
* **initialCellsValue**: a .val CD++ file is used to define the start values for each of the cells in the cell space.
  + 1. **CD++ Toolkit File and Message Types**

There are three types of CD++ Toolkit files that are of importance to this paper, these are: the .ma file; the .val file; and the .log file. The main parameters that define the cell space are contained in the .ma file (Figure 4 is a snapshot taken from the .ma file from [7]). Next, initial values for the cells may be defined in several ways within the toolkit: using a default initial value (initialvalue); creating a file that contains a list of initial values for the cells (initialCellsValue); or creating a “file that contains a map of values that will be used as the initial state for the cellular model”[6]. In [7], the initial cell values were explicitly defined in a list within a.val file (Figure 6).



***Figure 6. Sample taken from a .val file***

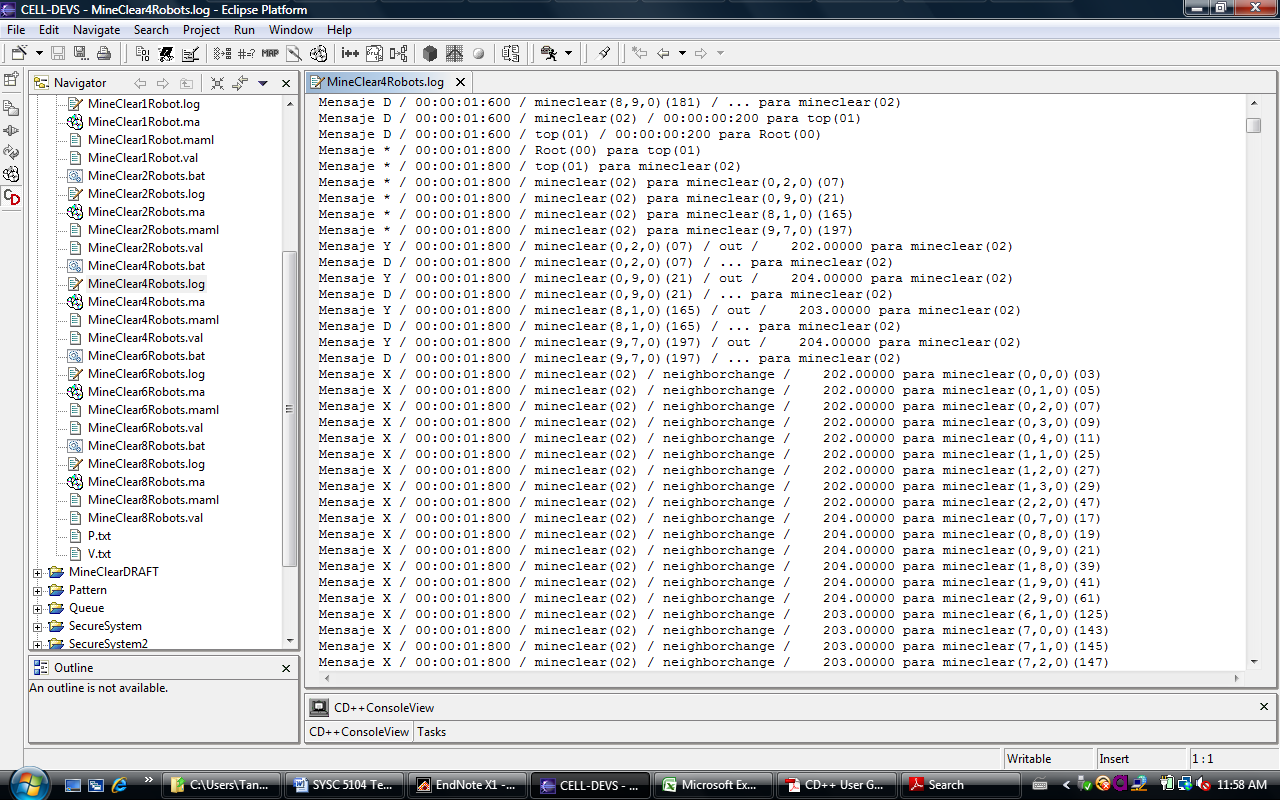
Finally, the .log file contains the resulting messages passed during the simulation of the model(s). There are several types of these messages, including:

* Message type I – Initialization message
* Message type \* – Internal message
* Message type X – External message
* Message type Y – Output message
* Message type D – Done message

It is the Y messages that are of particular import to the discussion related to visualization, as they provide the time, cell coordinates, and new state values with which to depict the results of the simulation. The complete format for the Y messages is as follows:

Message Y/time/cell/port/new state value

Figure 7 provides a snapshot of the .log file for one of the simulations from [7].



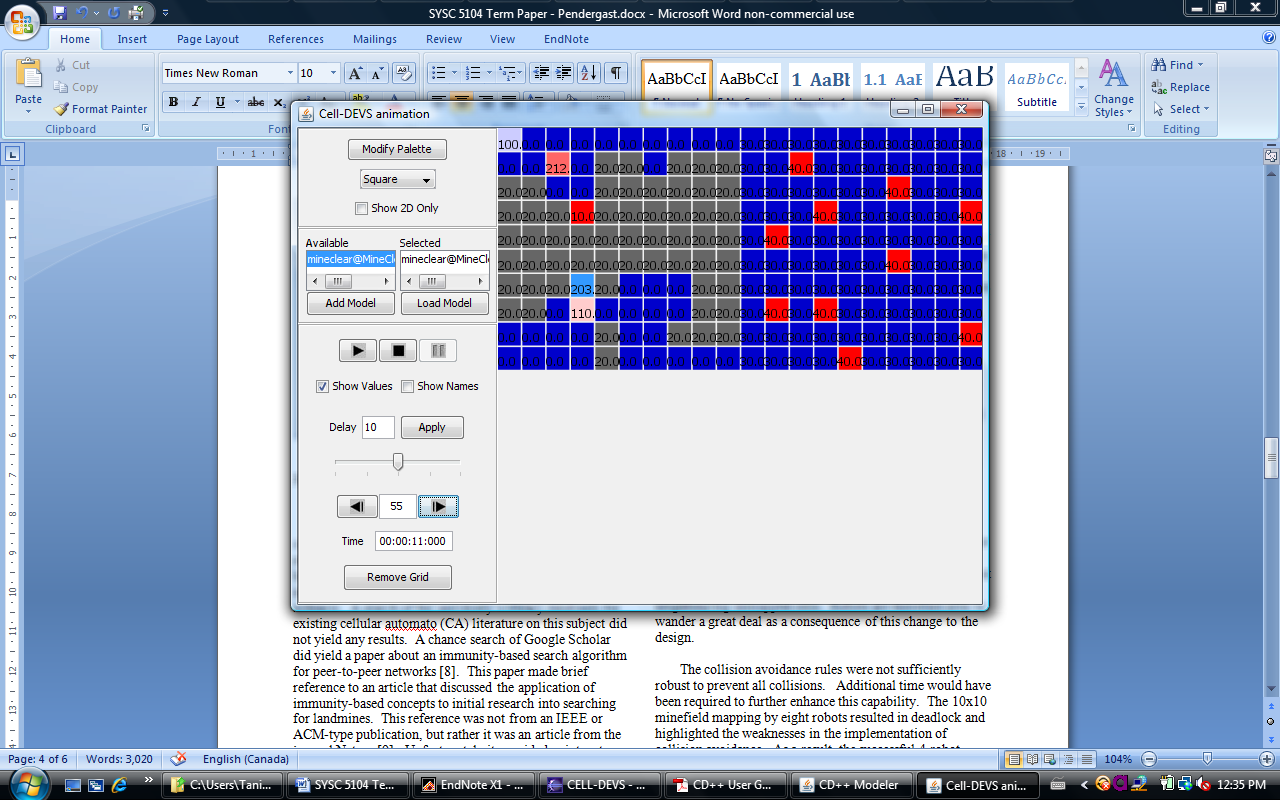
***Figure 7. Sample taken from a .log file***

* + 1. **CD++ Toolkit Modeler Tool**

The CD++ Toolkit also includes the Modeler tool that

allows for 2D visualization of simulations. The Modeler tool can be used to display only the top layer of the cell space, or it can be used to display each of the layers of the cell space. The tool can be used to define an extensive palette of colours with which to represent the defined state values. Multiple models can be loaded into the Modeler tool and animated at the same time.

The following figure depicts a screen capture from the simulation from [7], where the Modeler tool has been used to display both layers of the cell space. The two layers are displayed immediately adjacent to one another. The red line has been added for ease of viewing.



***Figure 8. Sample of 2D visualization from CD++ Modeler***

While the Modeler tool permits 2D visualization of simulations, it does not allow for 3D visualization. As a result, the use of other visualization and animation tools is required. Section 8 provides a discussion of a particular example of this: the use of Blender to produce a 3D visualization of a specific CD++ Cell-DEVS simulation.

1. **BACKGROUND TO THE MODELING AND SIMULATION OF A MINEFIELD MAPPING SCENARIO**

In [7], the system that was modeled was that of a minefield being scanned by robots in order to generate a map depicting the location of landmines (hereafter referred to as mines). This idea stemmed from my desire to model and simulate a minefield mapping or clearing-type of scenario. A search of the university’s library databases for existing cellular automato (CA) literature on this subject did not yield any results. A chance search of Google Scholar did yield a paper about an immunity-based search algorithm for peer-to-peer networks [8]. This paper made brief reference to an article that discussed the application of immunity-based concepts to initial research into searching for mines. This reference was not from an IEEE or ACM-type publication, but rather it was an article from the journal Nature [9]. Unfortunately it provided pointers to rather complex and non-cell space literature [10-12], so it was not helpful in identifying previous CA work in this area. It did, however, highlight a key point that was instrumental in defining the conceptual design for the CD++ Cell-DEVS model:

“...robots initially patrol the minefield randomly. When one discovers an area with a concentration of mines, it broadcasts a signal to the other robots, some of which migrate to the area...Only those in the vicinity of the signalling robot are recruited, while farther away robots continue to monitor other parts of the minefield. This simple mechanism easily outperforms traditional ‘raster’ scans, which divide a search area into a grid and comb it cell-by-cell, row-by-row.” [9]

The conceptual design for the CD++ Cell-DEVS implementation of a minefield mapping scenario stemmed from the aforementioned key point. Instead of designing a cell-by-cell and row-by-row type of scanning of the minefield, the design involved the random movement of robots about the minefield. The overarching concept was that a number of robots would be used to scan the ground of a minefield of a given size and generate a map depicting the location of the mines. To simplify development, the minefield was defined as a 10x10 grid. Individual models (each with their own CD++ implementation) were developed for one, two, four, six and eight robots mapping the minefield. Simulations were run for each of these models.

The initial intent had been to design and instantiate both collision avoidance and intelligent search capabilities for the robots. Intelligent search was a term I coined that meant providing the robots with the ability to seek out, and move into, unmapped cells before moving into already mapped cells. This was based on the notion that all of the robots were in communication with each other, or a central server, and were updating a common map. Unfortunately the local transition function rules associated with this particular design proved too difficult to implement and troubleshoot in conjunction with collision avoidance. Intelligent search worked extremely well in a one-robot scenario, but did not scale well beyond that. As a result, the final implementation did not include intelligent search, it only instantiated a degree of a collision avoidance capability for the robots. They moved randomly about the minefield without intelligently searching for neighbouring unmapped cells, and were therefore seen to wander a great deal as a consequence of this change to the design.

The collision avoidance rules were not sufficiently robust to prevent all collisions. Additional time would have been required to further enhance this capability. The 10x10 minefield mapping by eight robots resulted in deadlock and highlighted the weaknesses in the implementation of collision avoidance. As a result, the successful 4 robot mapping scenario was selected for the visualization effort discussed in the following sections.

1. **CD++ CELL-DEVS MINEFIELD MAPPING SIMULATION**

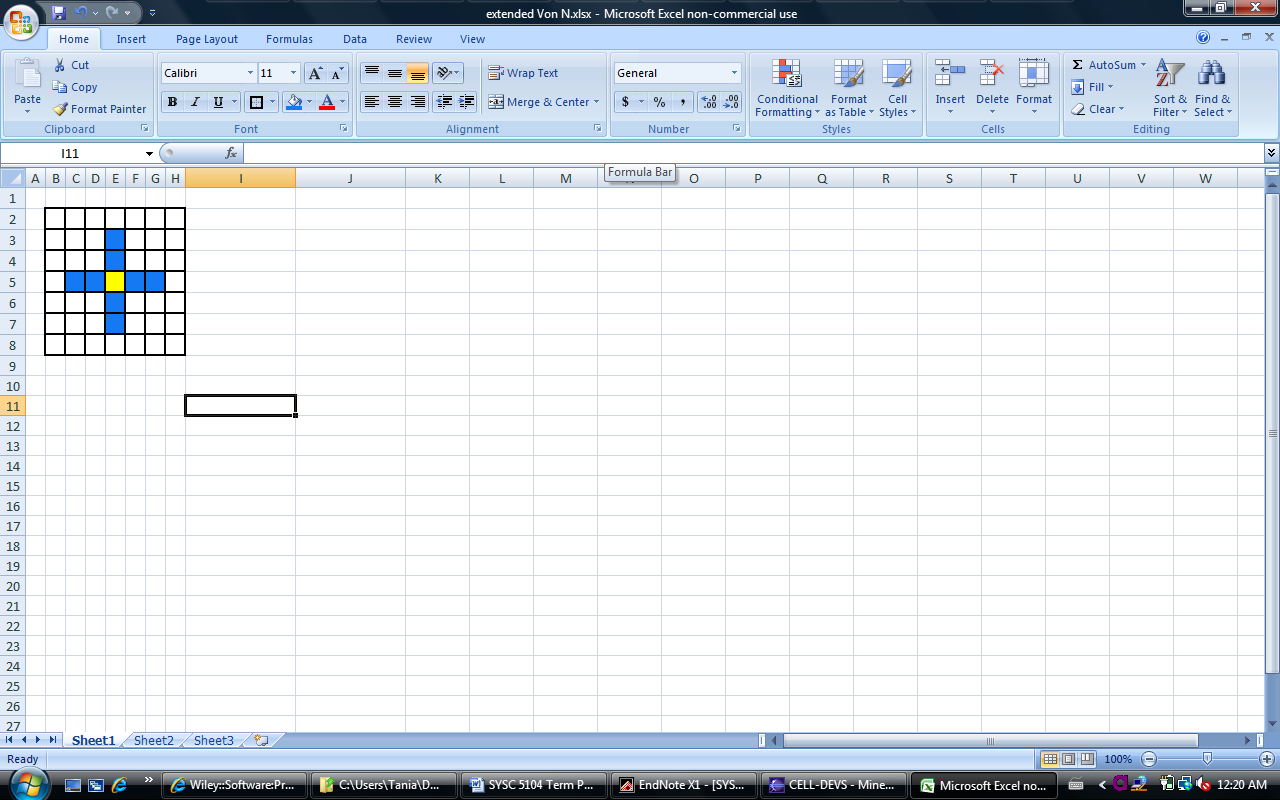
The cell space modeled in [7] represents a 10 cell x10 cell minefield. Each cell represents a realistic area of ground that a robot could be expected to scan for a mine. The minefield cell space has two layers. The first layer (layer 1) contains the actual disposition of the mines. The state values for this layer do not change during the simulation. The second layer (layer 0) contains the map of the minefield generated by the robots moving about the minefield. The state values for this layer change as the robots move about the minefield and scan the ground for mines. Table 1 depicts the state values that are defined for this simulation and the colour palette that was used for the CD++ Modeler visualization.

In this simulation [7], robots are constrained to move only to their North, East, South and West. For collision avoidance, the robots look to the two cells to their North, East, South, and West in order to avoid collisions with other robots. The neighbourhood may therefore be considered as a variation on the extended Von Neumann neighbourhood (which is meant to be a 5x5 rhombus).

***Table 1. State values and colour palette for CD++ simulation***

|  |  |  |
| --- | --- | --- |
| **Definition** | **State** | **Colour** |
| LAYER 1 – MINE DISPOSITION | | |
| No mine | 30 | Red |
| Mine | 40 | Blue |
| LAYER 0 – MINEFIELD MAP | | |
| Unmapped cell | 20 | Dk Grey |
| Unmapped cell with robot | 120 | Grey |
| Mapped cell – no mine | 0 | Blue |
| Mapped cell – mine | 10 | Red |
| Mapped cell – no mine, robot | 100 | Lt Blue |
| Mapped cell – mine, robot | 110 | Pink |
| Mapped cell – no mine, robot moving North | 201 | Med Blue |
| Mapped cell – no mine, robot moving East | 202 | Med Blue |
| Mapped cell – no mine, robot moving South | 203 | Med Blue |
| Mapped cell – no mine, robot moving West | 204 | Med Blue |
| Mapped cell – mine, robot moving North | 211 | Med Red |
| Mapped cell – mine, robot moving East | 212 | Med Red |
| Mapped cell – mine, robot moving South | 213 | Med Red |
| Mapped cell – mine, robot moving West | 214 | Med Red |

The variation on the extended Von Neumann neighbourhood is shown in Figure 9.



***Figure 9. Variation on Extended Von Neumann Neighbourhood***

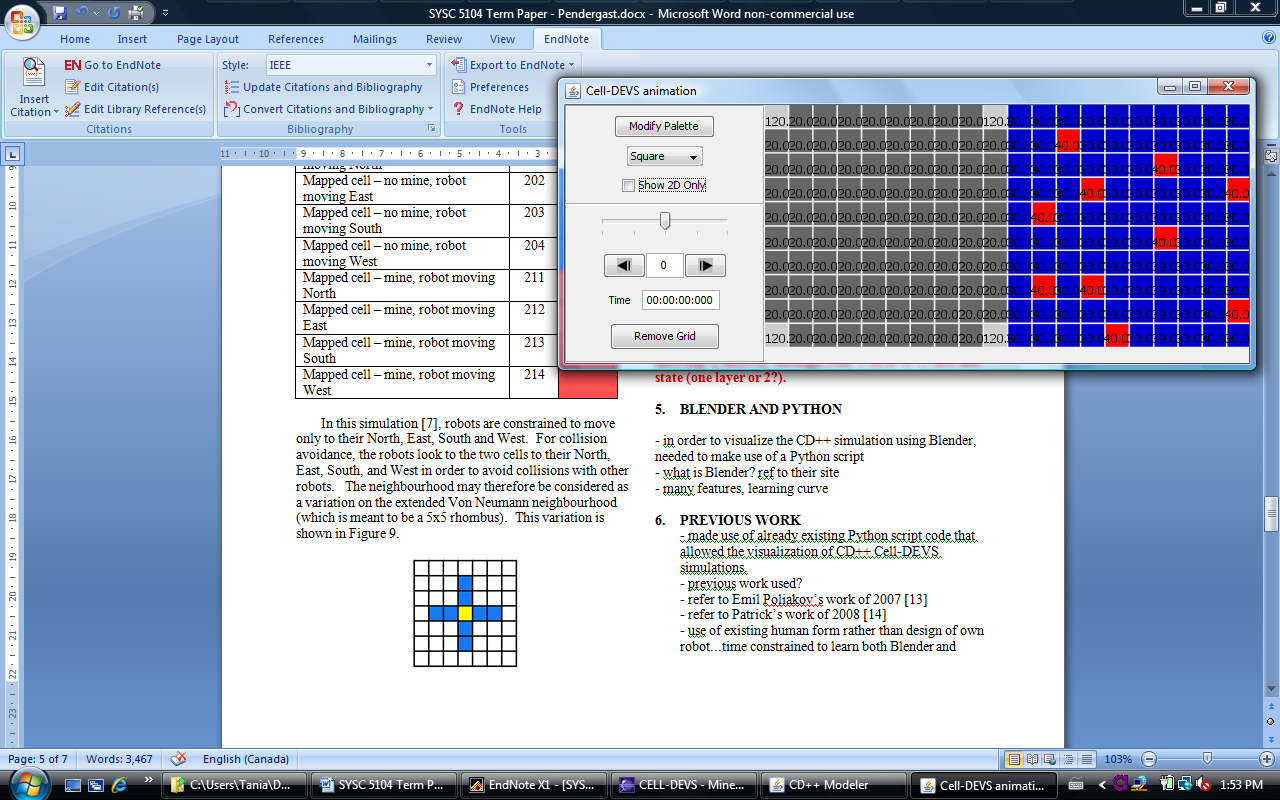
The minefield cell space was designed to have a border that constrains the movement of the robots to that space only. Therefore (as shown in Figure 4), in CD++ the border is defined as:

border : nowrapped

In order to model the appropriate behaviour of robots that are constrained to move only within the defined minefield cell space, unique zones were defined. Zones were defined for the four minefield borders and also for each of the four corners. Each zone has its own unique local transition function rules defined in order to properly model the behaviour expected from the cells within those zones. The definition of the zones for [7] is depicted in Figure 4.

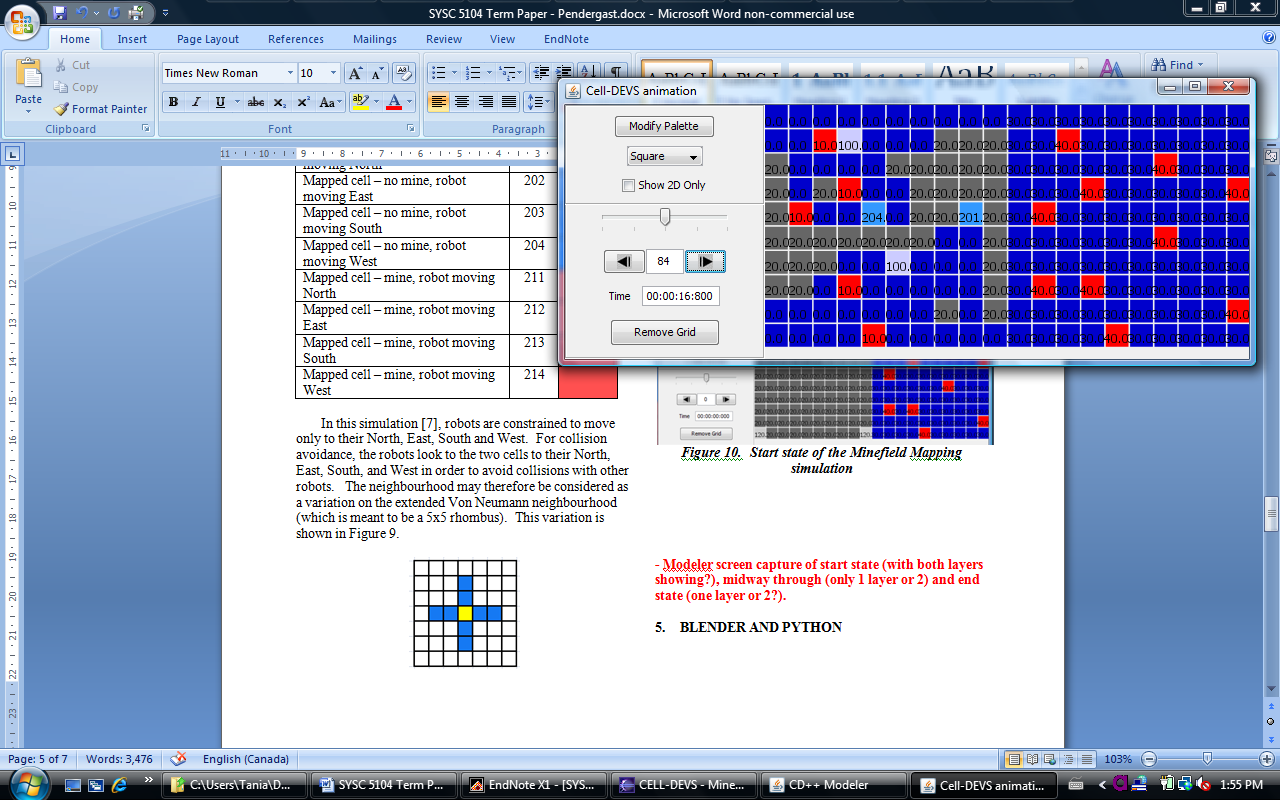
In order to begin the simulation with a defined placement of robots and the actual disposition of the mines within the minefield, a .val file was used. This file was used to list the initial state values for each cell within the cell space (both layers 0 and 1). A sample of the .val file for [7] is shown in Figure 6.

To demonstrate that the model appropriately captures the desired behaviour, the following three figures depict screen captures from the CD++ Modeler at different times throughout the simulation. The first figure (Figure 10) shows that the initial placement of the robots has been carried out in accordance with the .val file (ie. that a robot was placed in each of the four corners). Layer 0 is shown at left and layer 1 is shown at right.



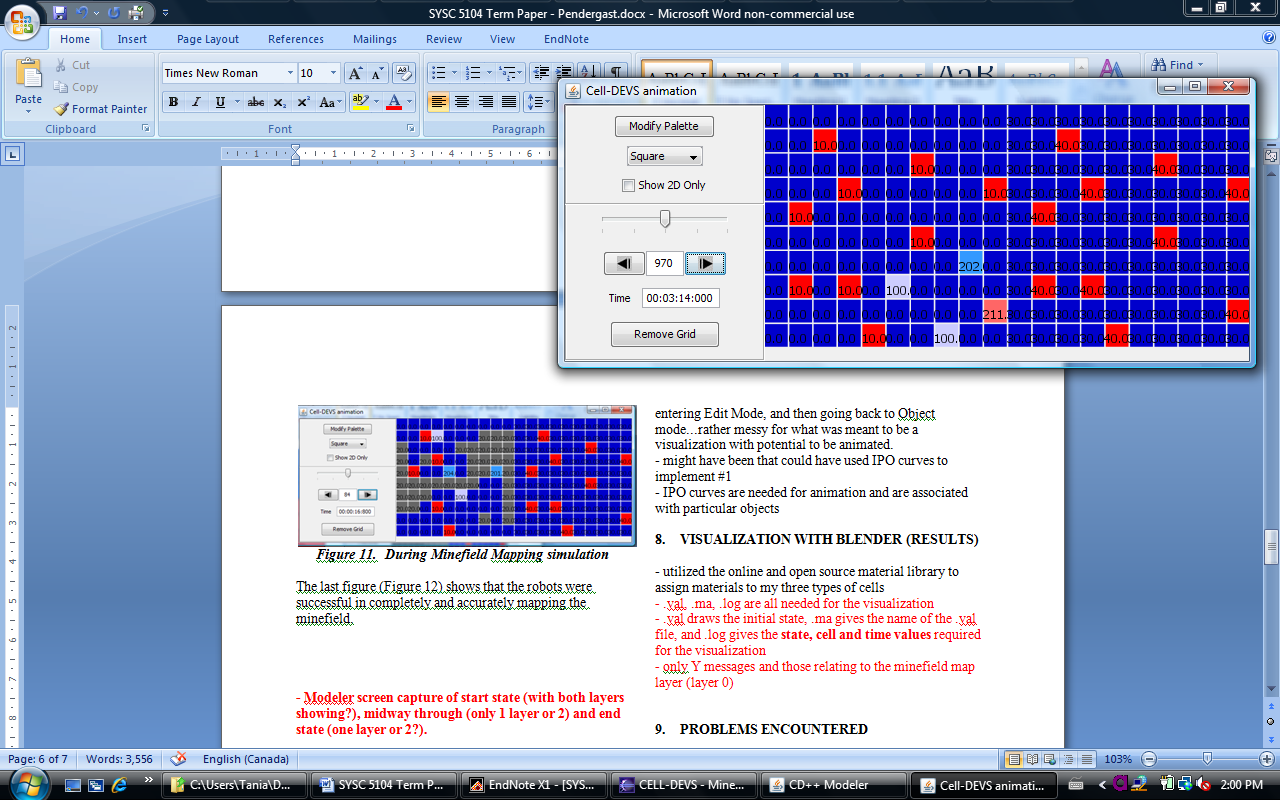
***Figure 10. Start state of the Minefield Mapping simulation***

The second figure (Figure 11) shows that the robots are accurately mapping the minefield and updating the minefield map (layer 0). Note that the location of the mines in the minefield map (layer 0 at left) accurately reflects their actual location within the minefield (layer 1 at right).



***Figure 11. During Minefield Mapping simulation***

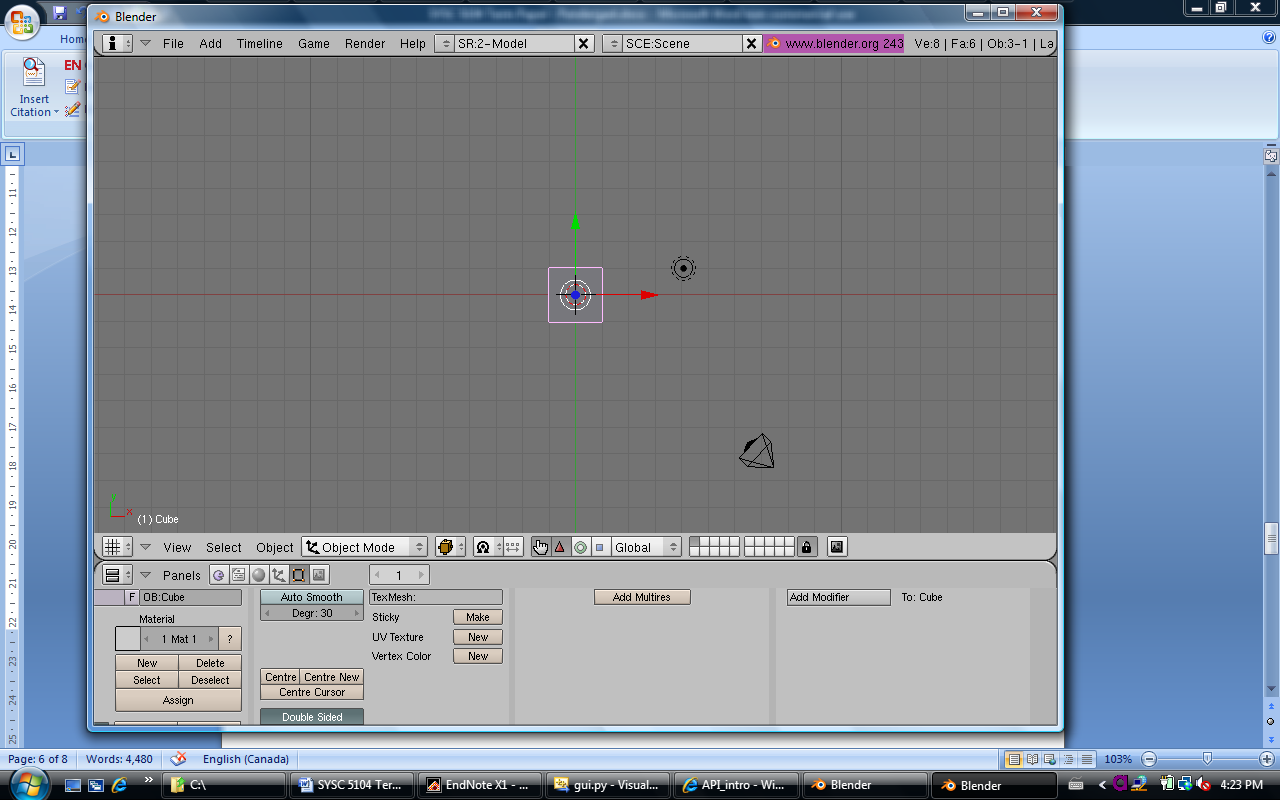
The last figure (Figure 12) shows that the robots were successful in completely and accurately mapping the minefield.



***Figure 12. End state of Minefield Mapping simulation***

1. **BLENDER AND PYTHON**

Blender is described as “an integrated application that enables the creation of a broad range of 2D and 3D content. [It] provides a broad spectrum of modeling, texturing, lighting, animation and video post-processing functionality in one package.” [13] It has a rather unique user interface and relies heavily upon keyboard shortcuts applied alone or in conjunction with the mouse. Figure 13 provides a snapshot of the Blender interface at startup. This screen capture shows the following key parts of the interface: the 3D View (1); the cube that is present at start up (2); the lamp that is present at start up (3); the camera that is present at start up (4); and the Buttons Window (5).

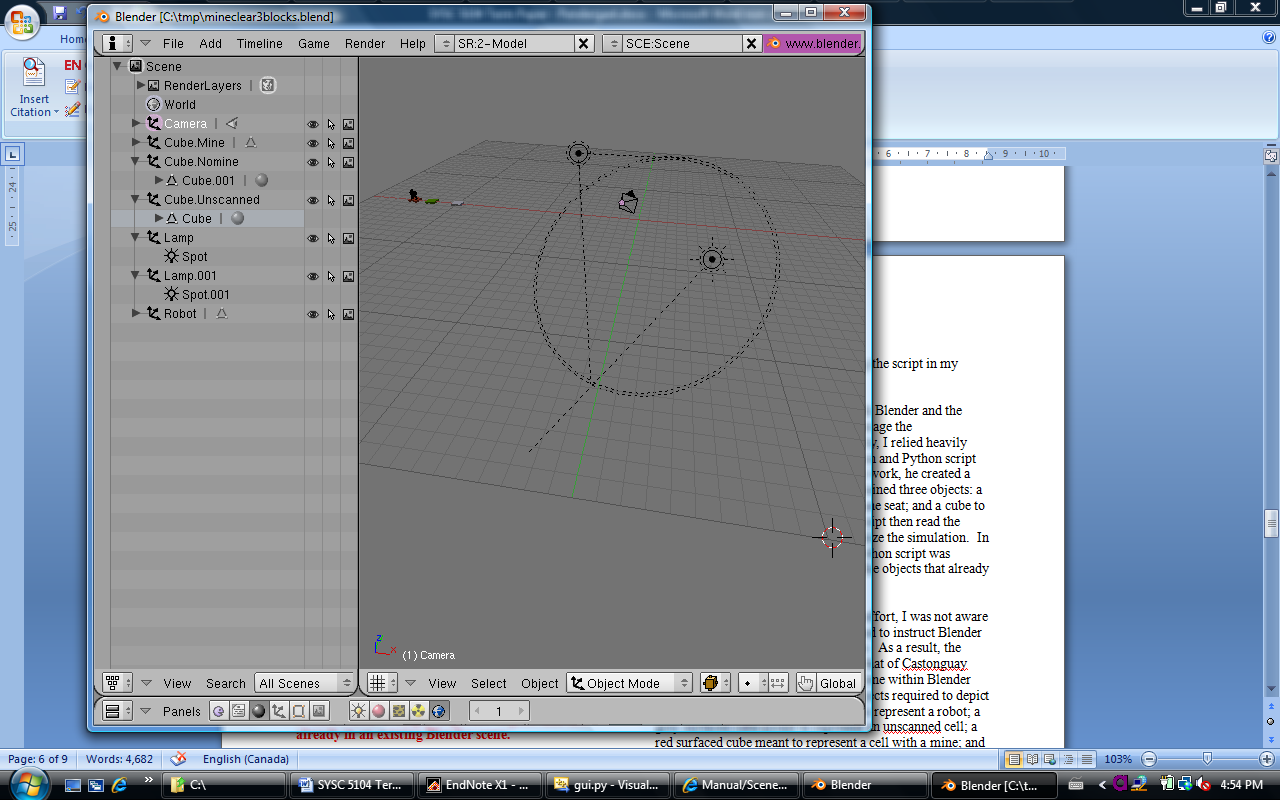


***Figure 13. Blender interface***

As a new user, I found that there was a rather steep learning curve associated with familiarizing myself with how to use Blender and how to harness all of its features. Within the time constraints associated with this particular project, I was most certainly not able to become a fully functional Blender user. The following section will discuss some of the issues associated with this fact.

The Blender site describes Python as a “general purpose scripting language” [13]. An ever expanding Blender Python API [14] exists that provides access to Blender’s internal functions in order to leverage them and also to extend Blender’s functionality [13, 14]. “The recommended version of Python is normally included and installed with the distribution of Blender. [13]” Due to the fact that Python was embedded in Blender, access to the Blender Python API modules must be made by running the scripts in Blender [14]. “You [cannot] import the Blender module into an external Python interpreter. [14]”

Within Blender, a scene is created to contain all of the objects that will form part of that scene. A single Blender file can contain multiple scenes, since it is “organized and set up to be able to contain an entire movie. [15]” Essentially a scene contains multiple objects that may then be organized into layers for ease of management [15]. The types of objects that can be present within a scene are constrained only by the imagination and skill of the scene’s designer. Each object is made up of multiple faces, vertices and edges, all of which can be manipulated using Blender. The scene will also include lamps to provide lighting for the scene. Lamps are required when a designer intends to animate the scene, if no lamp(s) are present then the rendered scene will be black. Cameras also form part of the scene. These can be placed throughout the scene to film it from various angles, and it is from a camera’s vantage point that an animation is filmed. The following figure   
(Figure 14) depicts the Outliner View within Blender, which provides a list of those objects that are found within the scene for the simulation from [7].

****

***Figure 14. Scene for Minefield Mapping visualization***

There are several modes in which to work within Blender. Of particular import to this visualization effort were the Edit and Object modes. The faces of the objects could only be selected in Edit Mode. So in order to apply different materials to an object it was necessary to be in Edit Mode. In Object Mode, objects could be moved, rotated or scaled. Edit Mode was important for performing more detailed changes to the geometry and materials of an object.

Only late in my familiarization with Blender was I able to begin to read about a capability called interpolation (IPO). I never did fully develop an understanding of the power that interpolation could lend to the visualization of my CD++ Cell-DEVS simulation. Had I known more about the power of interpolation, I believe that this would have had a significant impact upon my conceptual design. I did find that IPO “is the process of estimating an object’s position (or other attributes) based upon a known start and end value, and the time between the start and end [16]” and that it is used for the animation of a scene.

Animation software generally uses three methods to make 3D objects move, these are: key frames; motion curves; and paths. The Blender IPO system incorporates the first two, and either can be applied to objects in order to animate them. There are several types of interpolation within Blender, including: sequence; constraint; shape; texture; world; material; and object. Within each IPO type there are different types of channels against which interpolation (motion curves or key frames) can be applied. For example, within the Material IPO type, there are channels like R, G, B, and texR, texG, texB. In order to apply interpolation, one must first select a channel and then apply motion curves or key frames to it. Interpolation can be applied to more than one object (ie. one IPO can animate several objects). [17]

Python scripts can contain commands to create new scenes, objects, lamps, cameras, etc, or they can be used to simply duplicate objects that already exist in a Blender scene. They can also contain commands to associate IPO curves or key frames with a specific object or several objects.

1. **INFLUENCE OF PREVIOUS WORK**

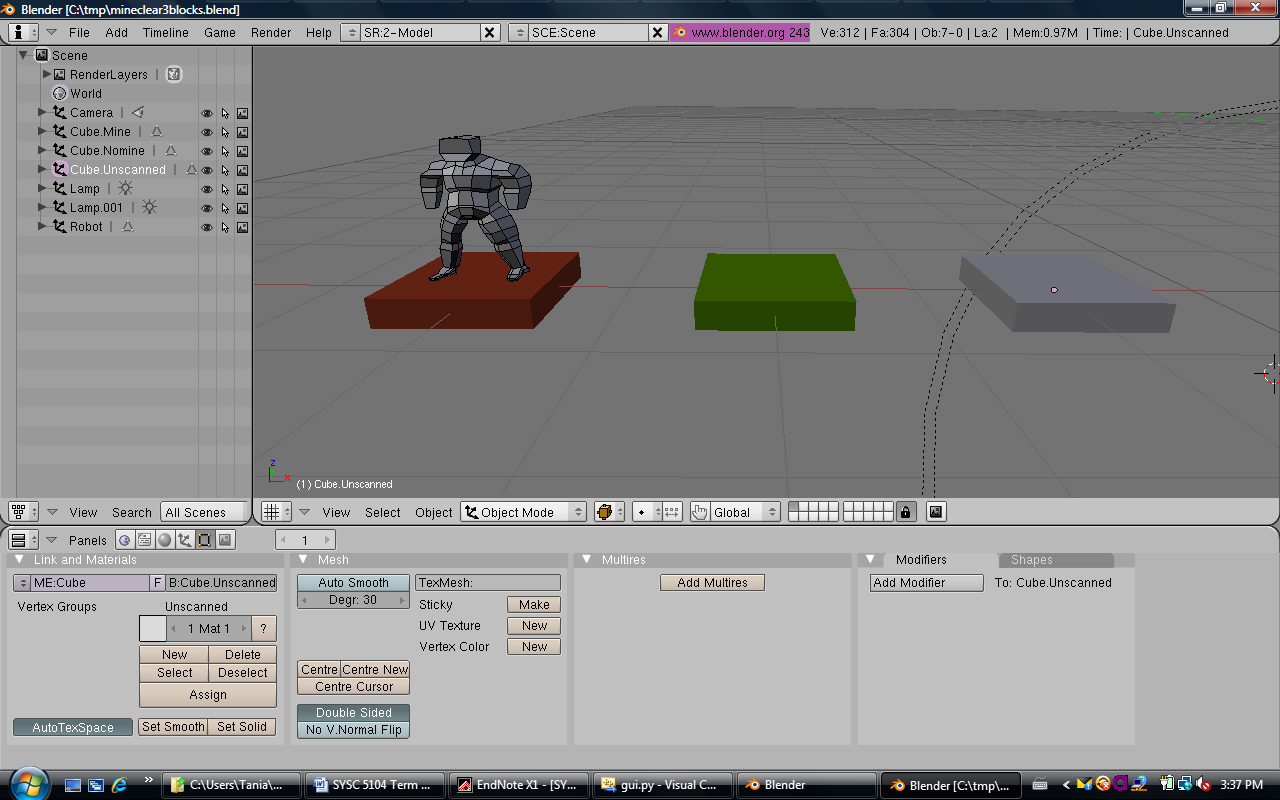
Blender was chosen as the tool to produce the 3D visualization of the CD++ Cell-DEVS minefield mapping simulation. This choice was predominantly based upon the fact that previous work had been done with respect to using Blender to interpret and visualize a CD++ Cell-DEVS simulation [18, 19].

In order to carry out 3D visualization of a CD++ simulation, a Python script is required. The original Python script, by Poliakov, enabled Blender to interpret a CD++ Cell-DEVS simulation [19]. This script was only designed to read and visualize the .log file from a CD++ Cell-DEVS simulation.

The original Python script was significantly improved by Castonguay in [18]. His version of the script added two new and important capabilities. The first of these was the ability to read the .ma file and search for the existence of a .val file. The script then read in the initial cell state values from the .val file and provided a 3D visualization of these in order to depict the simulation start state. The second capability that was added was that of a data log file for troubleshooting. The Python script created a data log text file that was used to capture key actions that were taken, including: the reading of the .ma file; the reading and visualization of the .val file (if applicable); and the reading and visualization of the .log file. This capability was particularly helpful for debugging the script in my particular case.

In order to familiarize myself with Blender and the existing Python script, and also to leverage the aforementioned additional functionality, I relied upon the airplane evacuation simulation and Python script developed by Castonguay [18]. In his work, he created a scene within Blender that already contained three objects: a human form; a cube to represent an airplane seat; and a cube to represent an airplane exit. The Python script then read the appropriate .val and .log files to visualize the simulation. In order to build the visualization, the Python script was structured to make Blender duplicate the objects that already existed in the scene.

At the outset of my visualization effort, I was not aware that the Python script could also be used to instruct Blender to create brand new scenes and objects. As a result, the approach that I took strongly mirrors that of Castonguay [18]. I eventually chose to create a scene within Blender that already contained the four key objects required to depict my simulation: a human form meant to represent a robot; a grey surfaced cube meant to represent an unscanned cell; a red surfaced cube meant to represent a cell with a mine; and a green grass surfaced cube meant to represent a cell without a mine. The logic behind my conceptual design is further explained in Section 7. The four objects in my initial Blender scene are shown in the following figure:



***Figure 15. Initial Blender scene for the Minefield Mapping simulation***

I was aware of the difficulties experienced by Castonguay in properly scaling an open source-obtained chair object to the scene. As a result, I chose to simply use the human form that both he and Poliakov utilized in order to depict my robots. The time constraints associated with completing this project did not permit me to explore the issues related to designing my own robot using Blender or properly scaling a different open source instantiation of a robot within my scene.

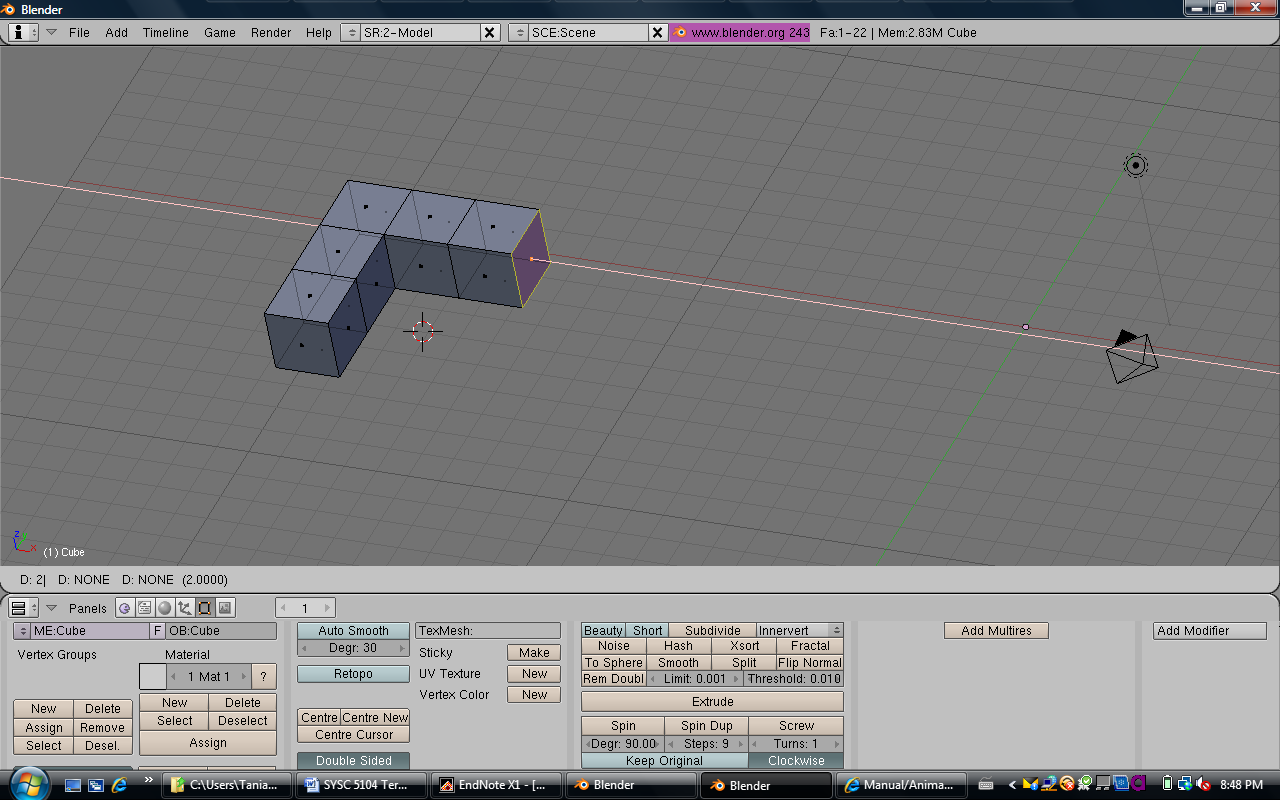
1. **BLENDER AND ITS IMPACT ON THE VISUALIZATION CONCEPTUAL DESIGN**

As previously mentioned, I faced a steep learning curve

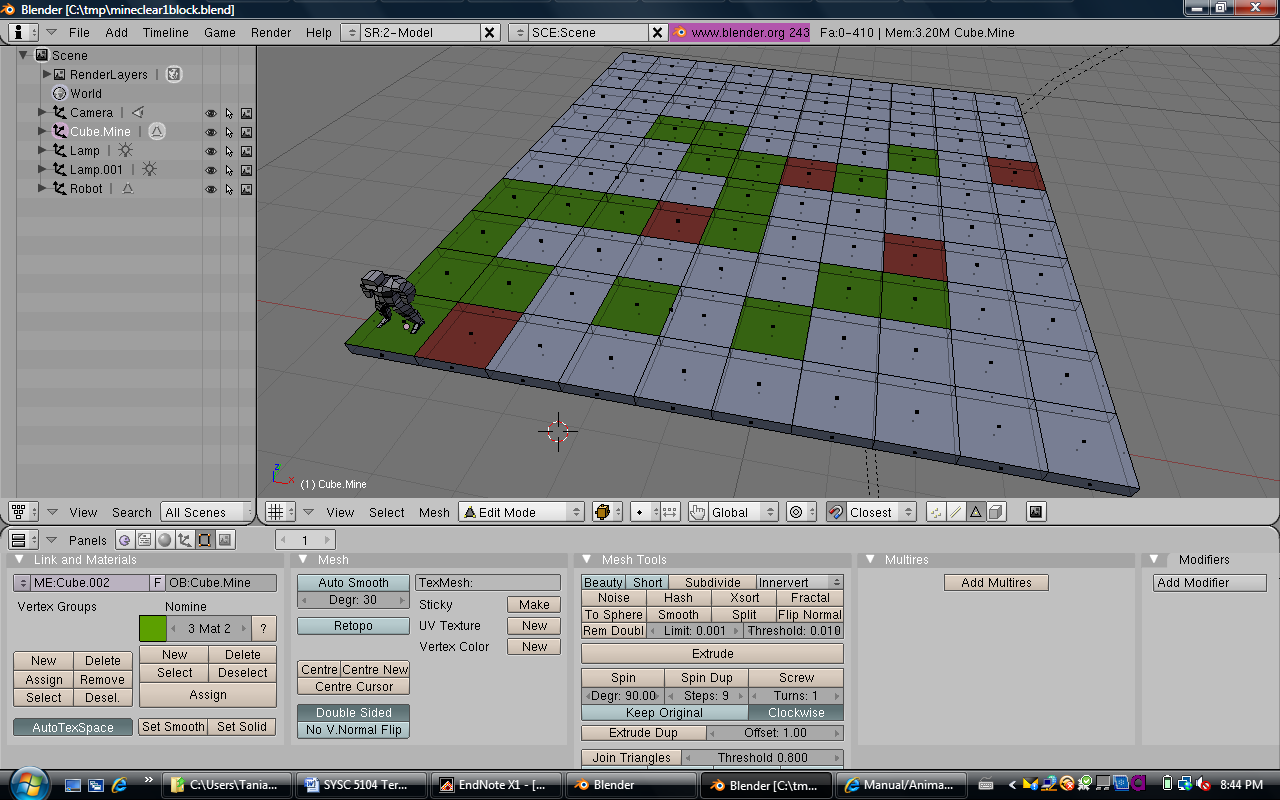
in familiarizing myself with Blender. Had I been more familiar the program, the conceptual design for the visualization of the CD++ Cell-DEVS simulation would likely have been better aligned to maximize Blender’s capabilities. Instead, I was forced to simultaneously learn how to use and harness Blender’s features and to develop my conceptual design. It was impossible to completely familiarize myself with the wide range of features and functionality provided by Blender in the time allocated for this project.

At the outset of developing the conceptual design for the visualization, I had hoped to take a somewhat different approach than Castonguay in terms of setting up the scene. I had thought that it would be more appropriate to design the scene in such a way that it was composed of only two key objects: the human form to represent the robot; and one cube whose top and bottom surfaces were divided into 100 separate faces (not to be confused with faces in the sense that a die has 6 faces, but rather Blender faces) to represent the cells of the minefield. This was made possible by first drawing a 1x1x0.2 cube and extruding its sides to form a 10x10x0.2 cube. In Blender, I found that different materials could be applied to each of these defined faces. As a result, the intent was to change a cell’s face once the robot had scanned the cell. Cells with mines would be changed assigned a red material and cells without mines would be assigned a green material. Unfortunately, I found that in order to apply new materials to a face, you had to be in Edit Mode. This did not align well with the intent to animate the simulation in the end.

The following two figures depict this initial conceptual design. Figure 16 shows a cube in the process of being extruded. Figure 17 depicts a cube that is 10x10x0.2 and has 100 faces on the top and bottom surfaces. Each face can be individually assigned a material, but only while in Edit Mode.



***Figure 16. Cube being extruded***



***Figure 17. Edit Mode view of a cube with 100 faces on top and bottom surfaces***

As a result of the aforementioned problem, and my lack of awareness of interpolation, I was forced to change my conceptual design. The final version of the conceptual design instead had three minefield squares (unscanned, mine, no mine) that were used to represent the minefield grid squares on the minefield map and a human form used to represent the robots (as shown in Figure 15). The visualization of the CD++ Cell-DEVS simulation was carried out by duplicating these objects to build the minefield and display the robots.

1. **VISUALIZATION WITH BLENDER (RESULTS)**

The key building blocks needed to complete the

visualization included: the CD++ Cell-DEVS .ma, .val, .log files from the successful 4 robot simulation; the Python script; and the Blender .blend file with the robot, unscanned cell, no mine cell, and mine cell. These will each be discussed in this section. The problems encountered during the visualization effort are outlined in the next section.

* 1. **CD++ Cell-DEVS Files**

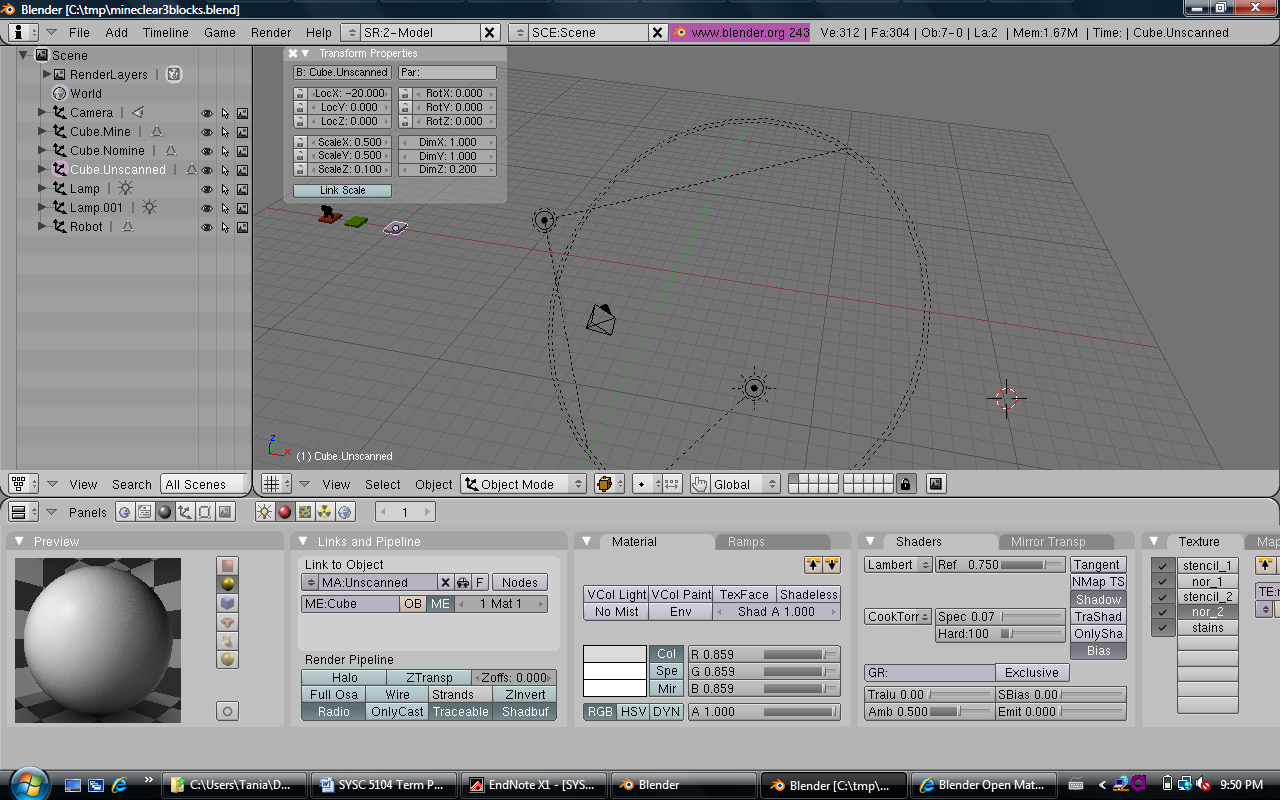
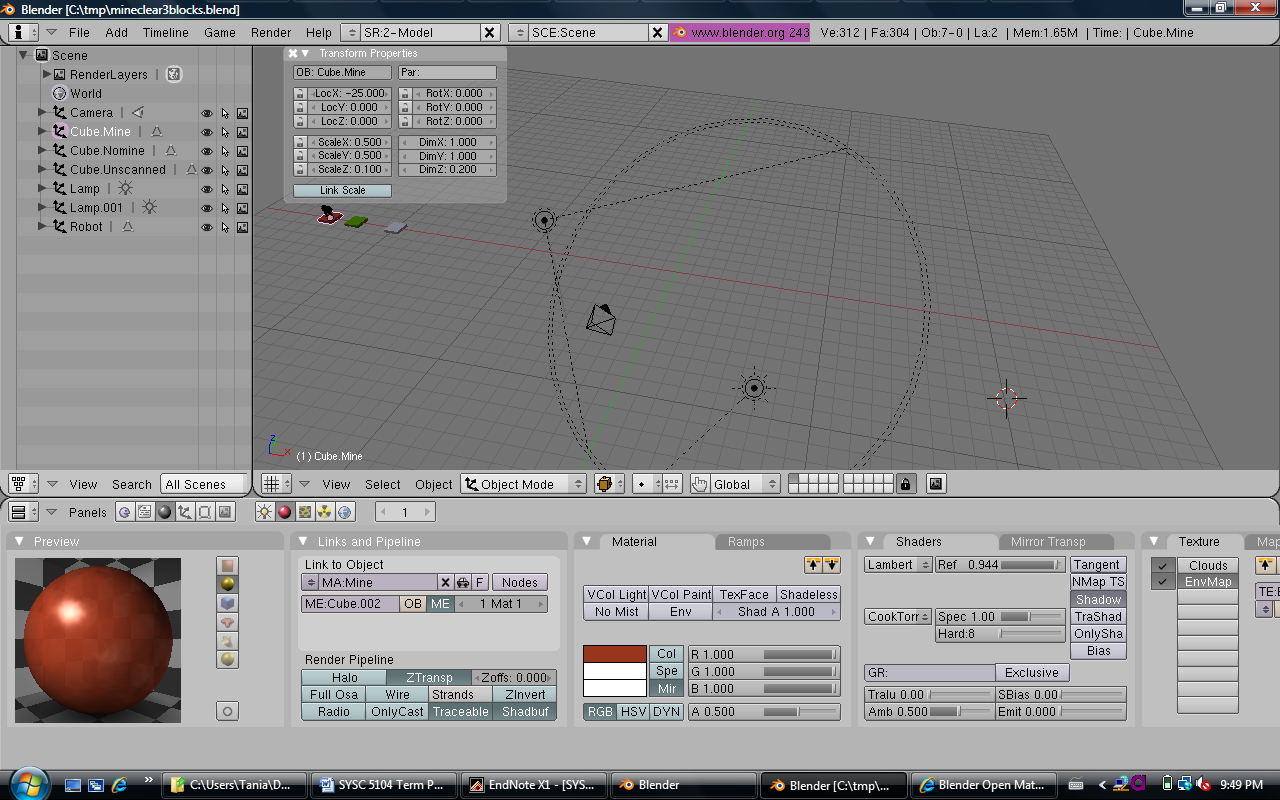
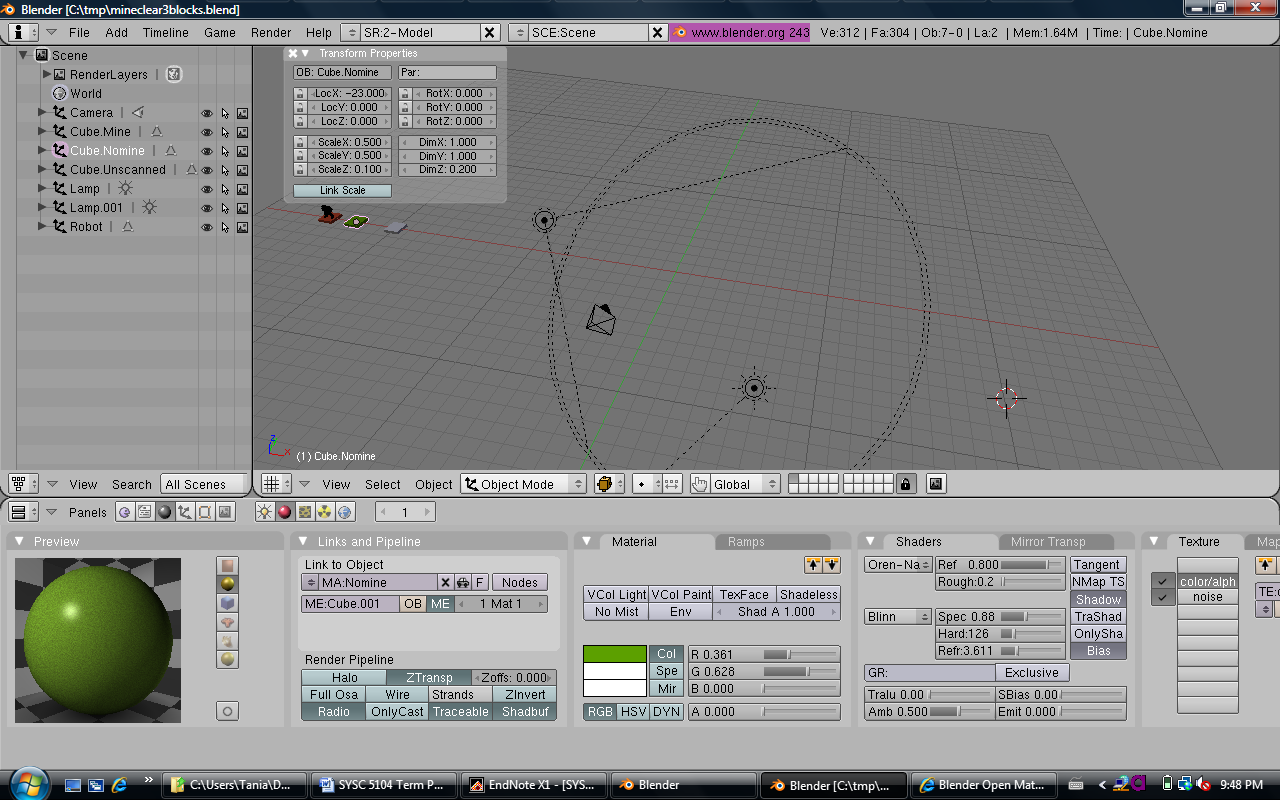
The .ma, .val, and .log files from CD++ provided the

data that was required to visualize the simulation of four robots scanning a 10x10 minefield. The .ma file provided the name of the .val file. The .val file listed the initial state values for each of the cells in both layers of the cell space. The .log file provided the Y messages from which the cell coordinates, state values and time were extracted.

* 1. **Blender .blend File**

The Blender .blend file was designed to contain the robot, the unscanned cell, the cell with a mine, and the cell with no mine. The visualization was then constructed by using the Python script to duplicate these objects in accordance with the cell coordinates, state values and times extracted from the CD++ files.

A distinguishing feature of this work in relation to that of Castonguay [18], is the application of materials to objects. While it is not readily apparent from the previous figures depicting screen captures from Blender, a realistic grass material was applied to the no mine cell, a ruby stone material was applied to the mine cell, and a grey wall material was applied to the unscanned cell (Figure 18). These materials were obtained from the Blender Open Material Repository [20]. The application of materials to an object is carried out while in Edit Mode in the 3D View and by making use of the Button Window’s Shading Panel. The Materials buttons within the Shading Panel were utilized to apply the materials to the specific objects. The screen shots in Figure 18 were taken from the Shading Panel’s Material preview.



***Figure 18. Green grass, ruby stone, and grey wall materials***

The final conceptual design involved the duplication of the four key objects in the initial scene. In the case of the robots, the initial scene’s robot was duplicated and positioned at each new location of a robot. At the robot’s previous location, the robot was removed from the cell. In the case of the minefield cells, all of the initial minefield cells were duplications of the grey unscanned cube in the initial scene. As cells were scanned by robots, the unscanned cells were removed and replaced by the appropriate (mine (red ruby) or no mine (green grass)) scanned cell. In order to carry out these tasks, it was important to understand how objects are removed from a scene in Blender. This was not as straightforward as it initially seemed.

In order to remove objects from a scene during a simulation, the objects must be “unlinked”. This is a rather interesting concept to understand. Through the troubleshooting phase of my design, I was at a loss to understand why the robots were replicating during the simulation. In the end, I determined that it was because the robots were not being cleared from their previous locations. This was tied directly to how objects are unlinked in Blender. While a simulation is running and an object is unlinked, its name still remains in memory. As a result, any subsequent duplications of the initial object will be given an extension to their object name (eg. Cube.Unscanned is unlinked, therefore the new duplicate will be called Cube.Unscanned.001). This becomes problematic when trying to find a particular object by name later in the simulation in order to unlink it. This problem and its resolution will be further discussed in Section 9.

* 1. **Python Script**

As previously discussed in Section 6, my Python script work primarily leveraged that of Castonguay [18]. It is important to note that his script leveraged the original work of Poliakov [19]. My Python script follows the structure of Castonguay’s work with major amendments to account for the nature of my particular simulation. The script contains the following key functions:

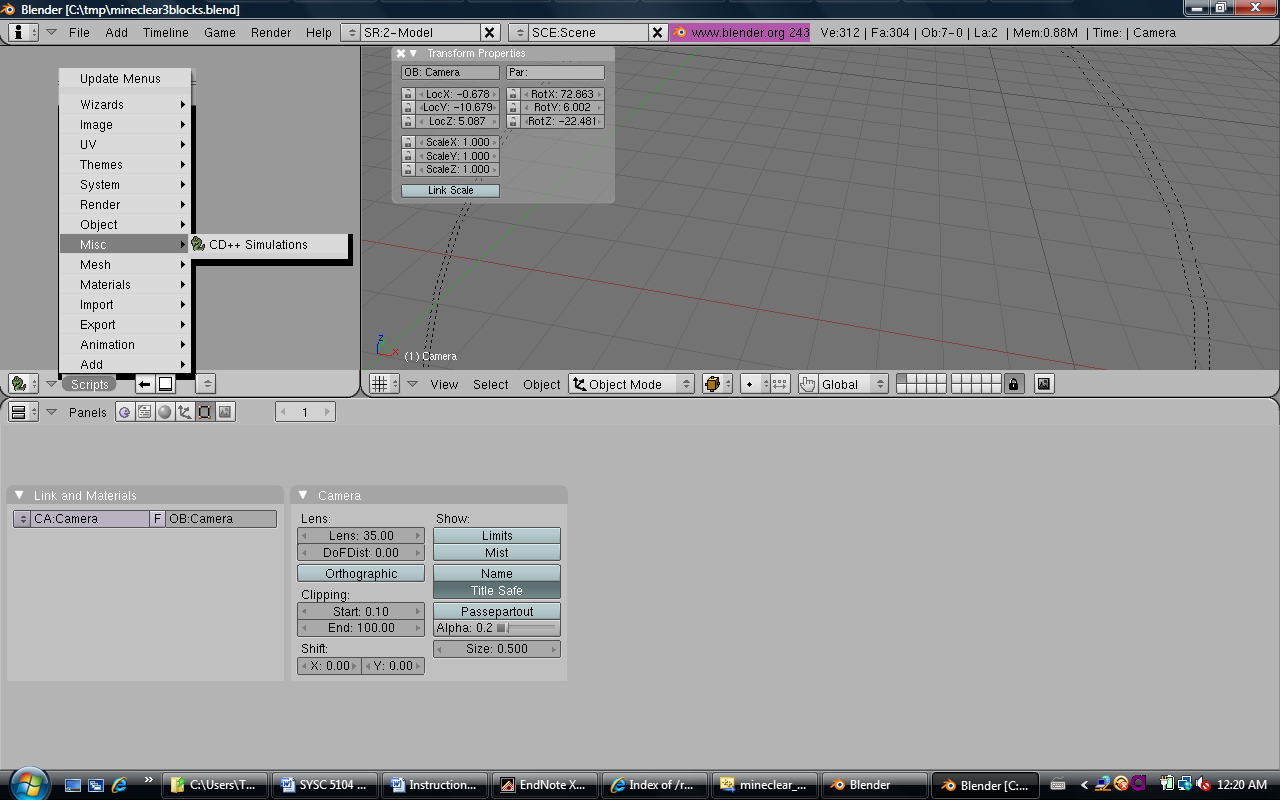
* read\_val, read\_ma, and read\_log
* apply\_log
* import\_maFile and import\_logFile
* gui
* buttonHandler

The first three functions are relatively self-explanatory. The read\_ma function searches the .ma file for either a default initial value to assign to all of the cells, or for a .val file that contains a list of initial values by cell. In the case of the minefield mapping simulation, this function would locate the MineClear4Robots.val file. The read\_val function takes the cell coordinates and initial state values (time = 00:00:00:000) from the .val file in order to visualize the initial state of the simulation. The read\_log function then takes the cell coordinates, state values and time values from the Y messages in the .log file in order to visualize the simulation from start to finish.

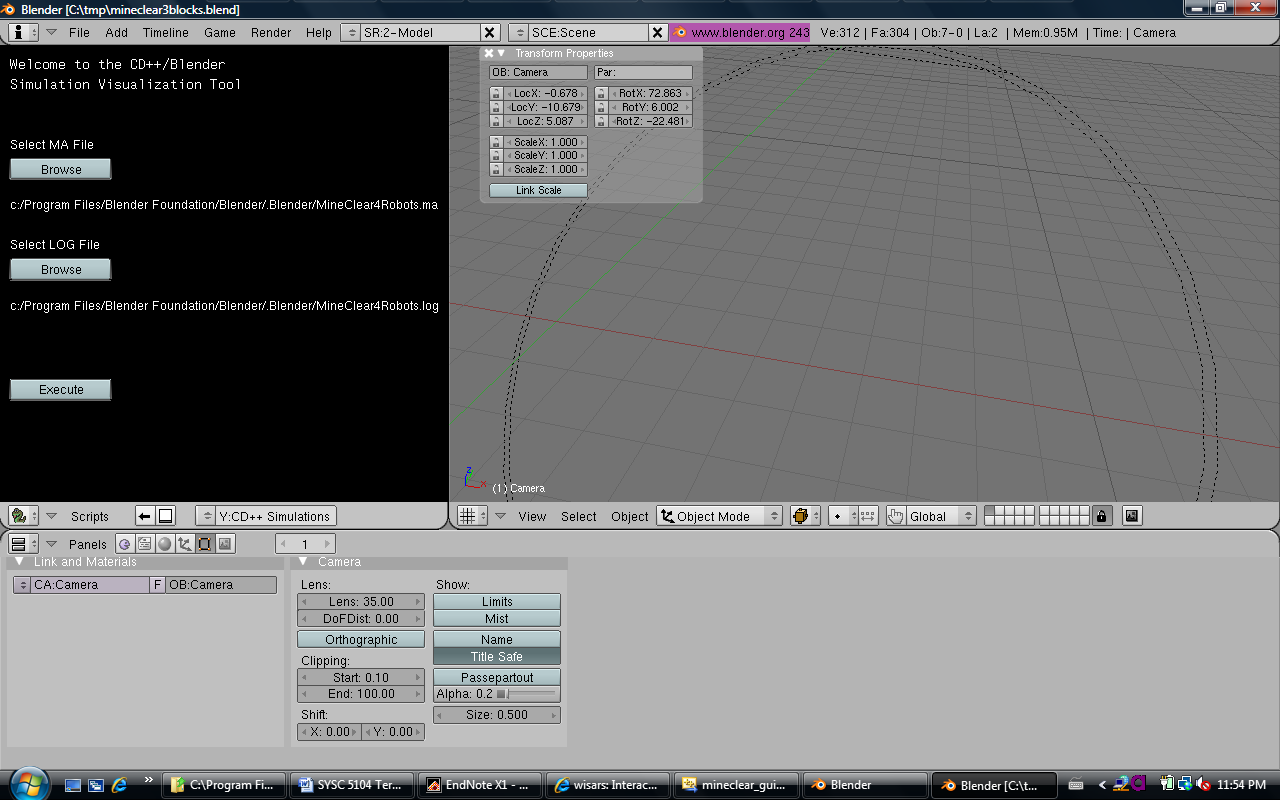
Both the read\_val and read\_log functions call the apply\_log function in order to carry out the visualization. An apply\_log function call is made each time a new state value is read in. The apply\_log function does many things. First, it pulls the value of the hours, minutes, seconds, milliseconds from the time value in order to calculate the total time in milliseconds. This value is used to set the current frame in Blender and associate the time, state and cell values with that frame. Secondly, it pulls the x, y, and z coordinates out of the cell value. Thirdly, it determines the state value (the Y message format for the state value has three decimal places, these are truncated here). Next, it creates two strings based upon the coordinates, one for the minefield cell name and one for the robot’s name. Lastly, the bulk of the apply\_log function deals with the handling of each of the state values when they are associated with a cell. In brief, the following actions are taken for the following state values:

* 20 (unmapped cell)
  + Get the Cube.Unscanned
  + Assign it a pointer (minefieldCell)
  + Link it to the scene, if it is not already
  + Select it
  + Duplicate it
  + Make this duplicate the active object
  + Name it according to its intended coordinates
  + Place it according to its coordinates
  + Deselect it
* 120 and time = 00:00:00:000 (robot on an unscanned cell at start)
  + Get the Robot
  + Assign it a pointer (robot)
  + Link it to the scene, if it is not already
  + Select it
  + Duplicate it
  + Make this duplicate the active object
  + Name it according to its intended coordinates
  + Place it according to its coordinates
  + Deselect it
  + Add an unscanned cell as for state = 20
* 120 and time != 00:00:00:000 (robot on an unscanned cell during the simulation)
  + Only add a new robot as per 120 and   
    time = 00:00:00:000
  + Do not add an unscanned cell as one already exists
* 0 and 10 (no mine in cell OR mine in cell: need to remove robot, cell already scanned and coloured accordingly)
  + Look for robot in the scene with coordinates matching the cell
  + Select it
  + Make it the active object
  + Rename it
  + Unlink it
* 100 and 110 (no mine and a robot OR mine and a robot: need to add a robot, cell already scanned and coloured accordingly)
  + Only add a new robot as per 120 and   
    time = 00:00:00:000
  + Do not add a scanned cell as one already exists
* 201-204 (cell just scanned, no mine: change the minefield cell to no mine)
  + Look for minefield cell in the scene with coordinates matching the cell
  + Select it
  + Make it the active object
  + Rename it
  + Unlink it
  + Get Cube.Nomine
  + Link it to the scene, if it is not already
  + Select it
  + Duplicate it
  + Make this duplicate the active object
  + Name it according to its intended coordinates
  + Place it according to its coordinates
  + Deselect it
* 211-214 (cell just scanned, mine: change the minefield cell to mine)
  + Look for minefield cell in the scene with coordinates matching the cell
  + Select it
  + Make it the active object
  + Rename it
  + Unlink it
  + Get Cube.Mine
  + Link it to the scene, if it is not already
  + Select it
  + Duplicate it
  + Make this duplicate the active object
  + Name it according to its intended coordinates
  + Place it according to its coordinates
  + Deselect it

The roles of the import\_maFile and import\_logFile functions are self explanatory. The gui function takes care of the display that is generated when the script entitle CD++ Simulations is selected from the Scripts menu under Misc (Figure 19) . The gui itself is shown in Figure 20.



***Figure 19. Selecting the CD++ Simulations script***



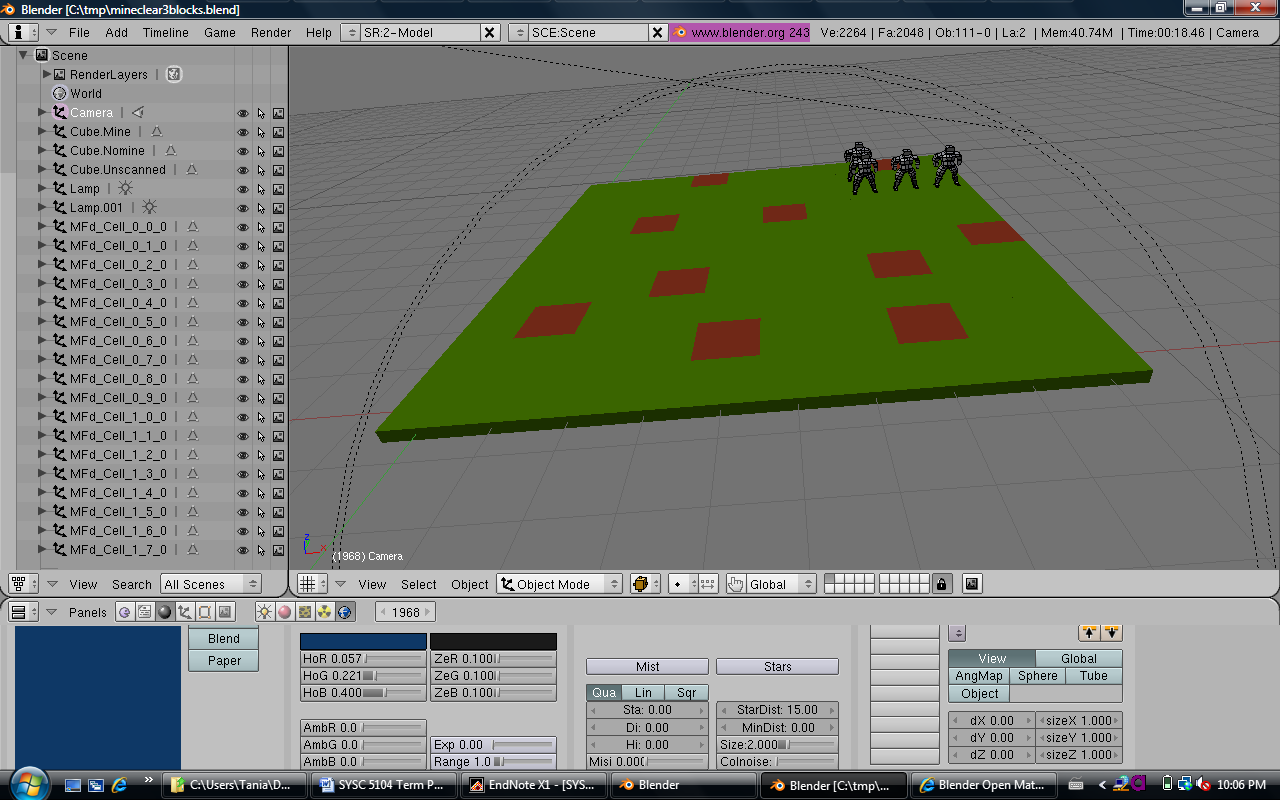
***Figure 20. CD++ Simulations script GUI***

* 1. **Visualization Set Up**

The following set up procedure should be followed to properly carry out the visualization of the C++ Cell-DEVS minefield mapping simulation:

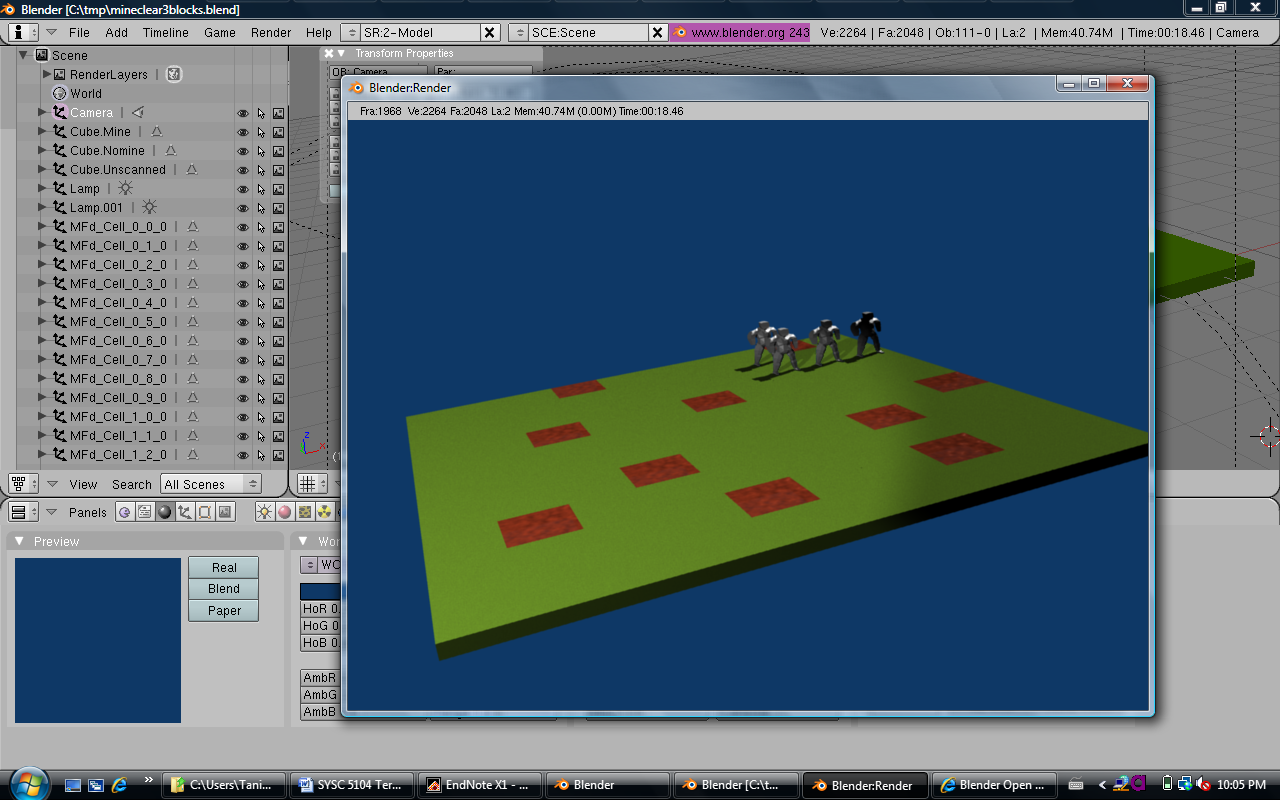
* Install Blender from [21]
* Install Python version 2.4 [22]
* Place **mineclear3blocks.blend** in   
  C:\Program Files\Blender Foundation\Blender\.blender
* Place **minefield\_gui\_v11.py** in   
  C:\Program Files\Blender Foundation\Blender\.blender\scripts.   
  *\*\*Delete any previous gui scripts from Castonguay [18] or Poliakov [19].*
* Place **MineClear4Robots.ma**, **MineClear4Robots.val**, and **MineClear4Robots.log** in   
  C:\Program Files\Blender Foundation\Blender\.blender
* Open mineclear3blocks.blend
* Load the CD++ Simulations script (see Figure 19)
* In the script GUI, select **MineClear4Robots.ma** and **MineClear4Robots.log** by browsing the .blender folder, if they are not already selected
* Hit the Execute button in the script GUI
* Watch the 3D visualization

The following figure (Figure 21) depicts the final frame of the minefield mapping visualization.



***Figure 21. Final frame of the visualization of the Minefield Mapping simulation***

The following figure (Figure 22) depicts the rendering of the final frame of the minefield mapping visualization. In order to animate the simulation, successive frames would need to be rendered. Since the conceptual design called for the duplication of objects, this prevents animation from being feasible (ie. there are not truly four robots, but rather multiple copies of the initial robot that are placed in the scene and then unlinked). Interpolation is required for animation, and it is associated with specific objects. In my case, the robots are shown to “move” by unlinking (deleting them) from their previous cell and duplicating them for the new cell. As a result, it is not possible to apply interpolation to a particular object in order to animate its movement. To do so would require changing the conceptual design, and as a result, the Python script. This problem will be further discussed in both Section 9 and 10.



***Figure 22. Rendering of final frame of the Minefield Mapping simulation***

1. **PROBLEMS ENCOUNTERED**
   1. **Unlinking Objects in Blender**

During the initial runs of the Python script, the robots were continually replicating throughout the minefield. It was not immediately apparent where the problem lay since the minefield was congested with robots. After an extended period of troubleshooting using both the data log .txt file and command line print commands, the cause finally became apparent. When a robot moved, the previous cell that it had occupied was not being cleared. This had a rather interesting, and negative, effect upon the ability to unlink objects.

At the start state of the simulation, a robot was placed in each of the four corners of the minefield (state value = 120, time = 00:00:00:000). In accordance with the apply\_log function in the Python script, each robot was named according to the cell that it occupied:

**Example.**

**For cell (0,0,0)**

robotName = ("Robot\_Cell\_%s\_%s\_%s"

%( xcoord,ycoord,zcoord))

**robotName becomes Robot\_Cell\_0\_0\_0**

Once this robot had scanned the cell, it would randomly pick a new direction in which to move and would then move. This resulted in two subsequent actions being taken:

* the previous cell was cleared by unlinking the robot with the name associated with that cell; and
* the new cell was occupied by duplicating the robot from the initial Blender scene, naming it according to the new cell’s coordinates, and placing it at the location associated with those cell coordinates.

This worked fine the first time around, since a robot was being unlinked from a particular cell for the first time. Unfortunately once a robot had been unlinked from the scene, the robotName associated with that cell could not be reused but it was still retained in memory. This created a problem in a roundabout type way. If a new robot entered that cell at a later time, it could not be given the name associated with that cell. Instead it was given a name with an extension (eg. Robot\_Cell\_0\_0\_0.001). As soon as this had occurred, this robot could not be unlinked from the cell in which it was located since the unlink command was being associated with the robot with that specific cell’s coordinates. The extension rendered this unlinking impossible.

While this may seem quite logical, it was not easy to identify this behaviour at the outset. Only after multiple troubleshooting efforts did this behaviour become apparent. By inserting .write commands within the apply\_log function I was able to identify the existence of objects with extensions. This behaviour was not something that had been purposely included in the Python script, so it was an indication of a problem that would need to be rectified. The resolution might seem quite simple at this time, however it did not immediately come to mind. In the end it was a fitting solution to fixing the unlinking problem. The robot being unlinked from a cell was renamed before it was unlinked. In so doing, the robot name associated with that cell could be reused again when another robot entered that cell at a later time.

The following example may serve to demonstrate the unlinking problem and its resolution:

**Example.**

1. Robot#1 is located in cell (0,0,0). It scans the cell. It then randomly picks a direction in which to move and then moves.
2. For this example, Robot#1 choses to move North to   
   cell (-1,0,0). The Robot#1 in cell (0,0,0) is unlinked and a new robot (Robot#2) is created in cell (-1,0,0).
3. Later, a new robot (Robot#3) moves into cell (0,0,0). It is named. Since the cell is already scanned, it randomly picks a new direction in which to move and then moves.
4. For this example, Robot#3 choses to move South to   
   cell (1,0,0). The Robot#3 in cell (0,0,0) is unlinked and a new robot (Robot#4) is created in cell (1,0,0).

|  |  |  |
| --- | --- | --- |
|  | **Problem Scenario** | **Resolution Scenario** |
| Robot#1 in  cell(0,0,0) | robotName = Robot\_Cell\_0\_0\_0 | robotName = Robot\_Cell\_0\_0\_0 |
| Robot#1 scans cell(0,0,0) then choses to move North to  cell(-1,0,0). | | |
| Robot#1 unlinked from cell(0,0,0) | \*unlink(robotName) | \*Rename robot to unlinked\_robot  \*unlink(unlinked\_robot) |
| Robot#2 created in  cell(-1,0,0) | robotName = Robot\_Cell\_-1\_0\_0 | robotName = Robot\_Cell\_-1\_0\_0 |
| New robot (Robot#3) moves into cell(0,0,0). Cell already scanned. | | |
| Robot#3 in cell(0,0,0) | robotName = Robot\_Cell\_0\_0\_0.**001** | robotName = Robot\_Cell\_0\_0\_0 |
| Robot#3 choses to move South to cell(1,0,0) | | |
| Robot#3 unlinked from cell(0,0,0) | **\*Cannot unlink.**  \*The robot with robotName = Robot\_Cell\_0\_0\_0 does not exist in the scene.  \*It does, however, exist in memory. So, all subsequent robots duplicated and placed in this cell will be have extensions added to their name. | \*Rename robot to unlinked\_robot  \*unlink(unlinked\_robot) |
| Robot#4 created in cell(1,0,0) | robotName = Robot\_Cell\_1\_0\_0 | robotName = Robot\_Cell\_1\_0\_0 |

* 1. **Inability to Animate**

The conceptual design for this effort involved the creation a Blender .blend file with an initial scene that contained the following objects: a human form representing a robot; a cube representing an unscanned cell; a cube representing a scanned cell with a mine; and a cube representing a scanned cell with no mine. The approach to making the robots move and the cells change colour was comprised of two steps: the duplication of the objects from the initial scene and their placement at the appropriate coordinates within the scene; and the unlinking of objects (ie. robot that had moved out of a cell, or an unscanned cell was removed and replaced by a scanned cell).

My understanding is that this conceptual design prevented the use of interpolation to animate the simulation. The reason for this is quite simple: interpolation is associated with an object or several objects that exist in a scene. This association is present for a defined period of time. The fact that my approach involved unlinking and creating new objects meant that interpolation could not be associated with any one object from the start of the simulation to the end in order to animate its movement. As a result, animation could not be achieved.

In order to resolve this issue, the Python code would need to be improved in two key ways:

* Only the required number of robots should be created and moved around the scene. Neither duplication nor unlinking should be used.
* Minefield cells should neither be duplicated nor unlinked, but rather their material should be changed to reflect the presence or lack of a mine.

In this way, interpolation could be applied to all of the objects within the scene from the start of the simulation to the end.

* 1. **Conceptual Design #1**

The initial conceptual design for this visualization effort involved creating a Blender .blend file with an initial scene that contained only two objects: a human form representing a robot; and a 10x10x0.2 cube representing the minefield. The 10x10 surface of the cube would have 100 individual faces that could each be assigned a material (Figure 17). This conceptual design was abandoned when it was determined that changing a face’s material could only be carried out in Edit Mode. I believed that this fact would prevent me from later being able to animate the scene.

It was only late in the development of conceptual design #2 that I started to become acquainted with interpolation. I was only able to learn so much in the limited time remaining, however I believe that interpolation could potentially have been used to change the materials assigned to each face. This approach would have enabled me to pursue the instantiation of conceptual design #1, and would have alleviated the need to duplicate and unlink individual cubes representing minefield cells. This could potentially have resolved the second of the key hurdles to animation discussed in Section 9.2.

* 1. **Vista**

The development efforts for this project were carried out on a laptop running Windows Vista. This was problematic in two ways. First, in the Castonguay Python script [18] the .txt file was created in the C:\Program Files\Blender Foundation\Blender\.blender folder. When I first ran my amended Python script, this .txt file was not created. I later determined that Vista was preventing me from creating this file within the Blender program folders. The path for the creation of the .txt file was later changed to c:\tmp to resolve this issue.

The second problem introduced by Vista was in relation to the .blend file. In Castonguay’s work, he inserted the .blend file in C:\Program Files\Blender Foundation\Blender\.blender folder and opened it from there. This approach was suitable for handling the .blend file once it was complete. Unfortunately, this was not suitable for the development phase of the .blend file. Vista prevented me from making any changes to the .blend file once it was placed in the program folder. Again, this issue was resolved by carrying out my development efforts with the .blend file in a folder within my Documents folder.

While these two problems may seem readily apparent at this time, it was initially somewhat frustrating to pinpoint the nature of the problem.

* 1. **Python .write Command**

The Python .write command was used to write key information to the data log .txt file in order to troubleshoot problems with the script and its behaviour. I had initially intended to use it predominantly to understand the behaviour of the apply\_log function, considering the fact that this was the function that was carrying out the visualization of the simulation. Unfortunately, the more .write commands I inserted in the apply\_log function the less the function would work. I finally identified that the .write command format used by Castonguay [18] was causing my Python script to create exceptions. The initial format used was as follows:

datalogfile.write("Processing:"+robotName+ "for logValue:"+logValue+"\n")

This problem was resolved by using the following format instead:

datalogfile.write("Processing: %s for %d\n" %(minefieldCellName,logValue))

* 1. **Tabs vs. Spaces in Python**

My initial lack of Python knowledge had a negative impact upon my ability to alter Castonguay’s code [18] to meet my specific simulation’s needs. Each time I removed sections of the script or altered it in some way, I would render that particular section useless. I later determined that this was associated with the fact that my IDE was mixing tabs with spaces and rendering the code unreadable. This is because in Python, indentation is used to group statements into blocks [23]. By mixing tabs and spaces I was negatively affecting the grouping of the statements in the script.

The resolution for this problem was quite simple: properly configure the IDE that is used. The IDE should be configured to either:

* Add spaces when pressing the tab key; OR
* Add tabs when pressing the tab key. The programmer should then also only use tabs for indentation. [23]

1. **FUTURE WORK**

While this visualization effort was an extremely valuable experience for the author, it could serve as a springboard for future work. The following sections discuss two overarching recommendations that should be combined in future work in this area.

* 1. **Design in Order to Animate**

The conceptual design incorporates the content of the initial Blender .blend file and also the structure and design of the Python script. Section 9.2 discussed two ways in which the Python script could be improved in order to allow for animation. Section 9.3 discussed the potential of pursuing conceptual design #1, which involved the use of the 10x10x0.2 cube in the initial Blender scene. As a result, my recommendations are that future work should:

* Avoid the use of object duplication and unlinking in the Python script;
* Create only the required number of robots and manage their movement appropriately within the scene (may need to add previous, current and next coordinates to manage this) using the Python script; and
* Investigate the option of using the 10x10x0.2 minefield with 100 faces on the 10x10 surface in the initial Blender scene, rather than the duplication and use of 100 separate cells.
  1. **Investigate the Use of Interpolation**

The limited exposure to interpolation during this work did not enable the author to fully appreciated the functionality that interpolation could provide. Two points were made clear from reading the Blender documentation:

* Interpolation is required for the animation of a scene; and
* Interpolation animates an object, or objects, by “estimating [their] position based upon a known start end value, and the time between the start and end [16]”.

By addressing the issues raised in Section 10.1, it may be that the primary focus could instead be upon how to apply interpolation to a scene. As a result, my recommendations are that future work should:

* Overcome the design hurdles early on;
* Shift focus to understanding and applying interpolation to the scene. Key things such as IPO types, channels, IPO curves, key frames, etc. should be understood;
* Investigate the use of IPO types, and their associated channels, in order to animate the scene;
* For the 10x10x0.2 cube case, investigate whether interpolation can be applied to change the material that is assigned to any of the 100 faces on the 10x10 surface without having to enter into Edit Mode.

1. **CONCLUSION**

This paper has presented the background, design and implementation steps, and problems encountered and resolved, with respect to an effort to visualize a CD++ Cell-DEVS simulation using both Blender and a Python script. It has also provided recommendations for future work. While the author has no other 3D software against which to compare functionality and ease of use, it would appear that Blender could be a fitting visualization tool for CD++ Cell-DEVS simulations. Those who pursue future work in this area can choose to leverage the existing Python script and the lessons learned from this work to develop their own unique scenarios, or they can expand upon the current Cell-DEVS minefield mapping scenario. The possibilities are many and the potential approaches are only limited by the imagination and time of future developers.

**References**

[1] Blender, "Blender," 2008, http://www.blender.org/.

[2] G. Wainer, "Introduction to the DEVS Modeling and Simulation Formalism," in *Discrete-Event Modeling and Simulation: A Practitioner's Approach (Draft)*, pp. 1-22.

[3] G. Wainer and N. Giambiasi, "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation," *Simulation,* vol. 71, No. 1, pp. 22-39, January 2001.

[4] G. Wainer, "The Cell-DEVS Formalism," in *Discrete-Event Modeling and Simulation: A Practitioner's Approach (Draft)*, pp. 1-22.

[5] G. Wainer, "CD++: A Toolkit to Develop DEVS Models," *Software - Practice and Experience,* vol. 32, No. 13, pp. 1261-1306, November 2002.

[6] G. Wainer, "Introduction to the CD++ Toolkit," in *Discrete-Event Modeling and Simulation: A Practitioner's Approach (Draft)*, pp. 1-32.

[7] T. Pendergast, "SYSC 5104 Assignment 2," Carleton University, 2008.

[8] N. Ganguly and A. Deutsch, "A Cellular Automata Model for Immune Based Search Algorithm," in *Proceedings of the 6th International Conference on Cellular Automata for Research and Industry*, Amsterdam, Netherlands, 2004, pp. 61-70.

[9] E. Klarreich, "Inspired by Immunity," *Nature,* vol. 415, No. pp. 468-470, 31 January, 2002.

[10] A. C. Morris, S. P. N. Singh, and S. M. Thayer, "Development of an Immunology-Based Multi-Robot Coordination Algorithm for Exploration and Mapping," Conference on Intelligent Robots and Systems, Presentation, October 2, 2002.

[11] S. P. N. Singh and S. M. Thayer, "ARMS: Autonomous Robots for Military Systems - A Survey of Collaborative Robotics Core Technologies and Their Military Applications," Carnegie Mellon University Robotics Institute, CMU-RI-TR-01-16, 2001.

[12] S. P. N. Singh and S. M. Thayer, "A Foundation for Kilorobotic Exploration," in *Proceedings of the 2002 Congress on Evolutionary Computation*, 2002.

[13] Blender, "Manual/Introduction - BlenderWiki," 2008, http://wiki.blender.org/index.php/Manual/Introduction.

[14] The Blender Python Team, "Module API\_Intro: The Blender Python API Reference: Built Thu May 10 20:32:02 EST 2007," 2007, http://www.blender.org/documentation/245PythonDoc/.

[15] Blender, "Scene Management Structure," 2008, http://wiki.blender.org/index.php/Manual/Scene\_Management.

[16] Blender, "Manual/Ipo Types," 2008, http://wiki.blender.org/index.php/Manual/Ipo\_Types.

[17] Blender, "Manual/Animation Basics," 2008, http://wiki.blender.org/index.php/Manual/Animation\_Basics.

[18] P. Castonguay, "CD++ Simulations version 1.0.2," 2008.

[19] E. Poliakov, "CD++ Simulations version 1.0.1," 2007.

[20] Blender, "Open Material Repository," 2008, http://www.blender-materials.org/index.php.

[21] Blender, "Blender 2.43 Download," 2007, http://download.blender.org/release/Blender2.43/blender-2.43-windows.exe.

[22] Python, "Python version 2.4 - Download," 2008, http://www.python.org/ftp/python/2.4.4/python-2.4.4.msi.

[23] Python, "How to Edit Python Code," 2008, http://wiki.python.org/moin/HowToEditPythonCode.