CSOUND CONFERENCE 2011

Csound and Object-orientation : Designing and implementing Cellular Automata for Algorithmic Composition

Reza Payami
reza_payami@yahoo.com

Introduction

Although object-oriented design [2] and implementation [3] may not seem to be necessary when working with computer music languages and environments such as Csound, it may help having more strong-typing facilities and well-structured reusable opcode libraries. Moreover, an object-oriented design would help solving a problem through a point of view, closer to how it really exists in the real world.

In this article the problem of designing and implementing the cellular automata[5-9], as an algorithmic composition component used by some composers like Xenakis[2] (Horos (1986)) or Synthesizers like Chaosynth[6], and its two frequently-used examples, "Game of Life"[5, 6] and "Demon Cyclic Space"[5] would be detailed. Assuming that Csound provided us with some object-oriented programming facilities [3], the alternative approach would be compared with the currently existing one. This article could also be used as a basis for a workshop. The alternative C++ opcodes implementation could also be explained in a corresponding workshop.

I. Cellular Automata

A cellular automaton, or simply CA, consists of:

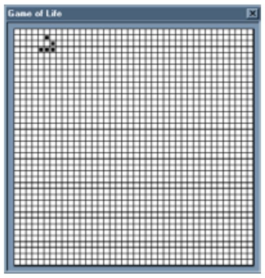
- A matrix, or grid, of cells, each of which can be in one of a finite number of states
- A rule that defines how the states of the cells are updated over time

The matrix of cells can have any number of dimensions. Given the state of a cell and its neighbors at time *t*, the rule determines the cell state at time *t + 1*. This will become clearer after looking at the following some concrete examples. The first generation and the corresponding states could be initially set using random numbers.

Game of Life

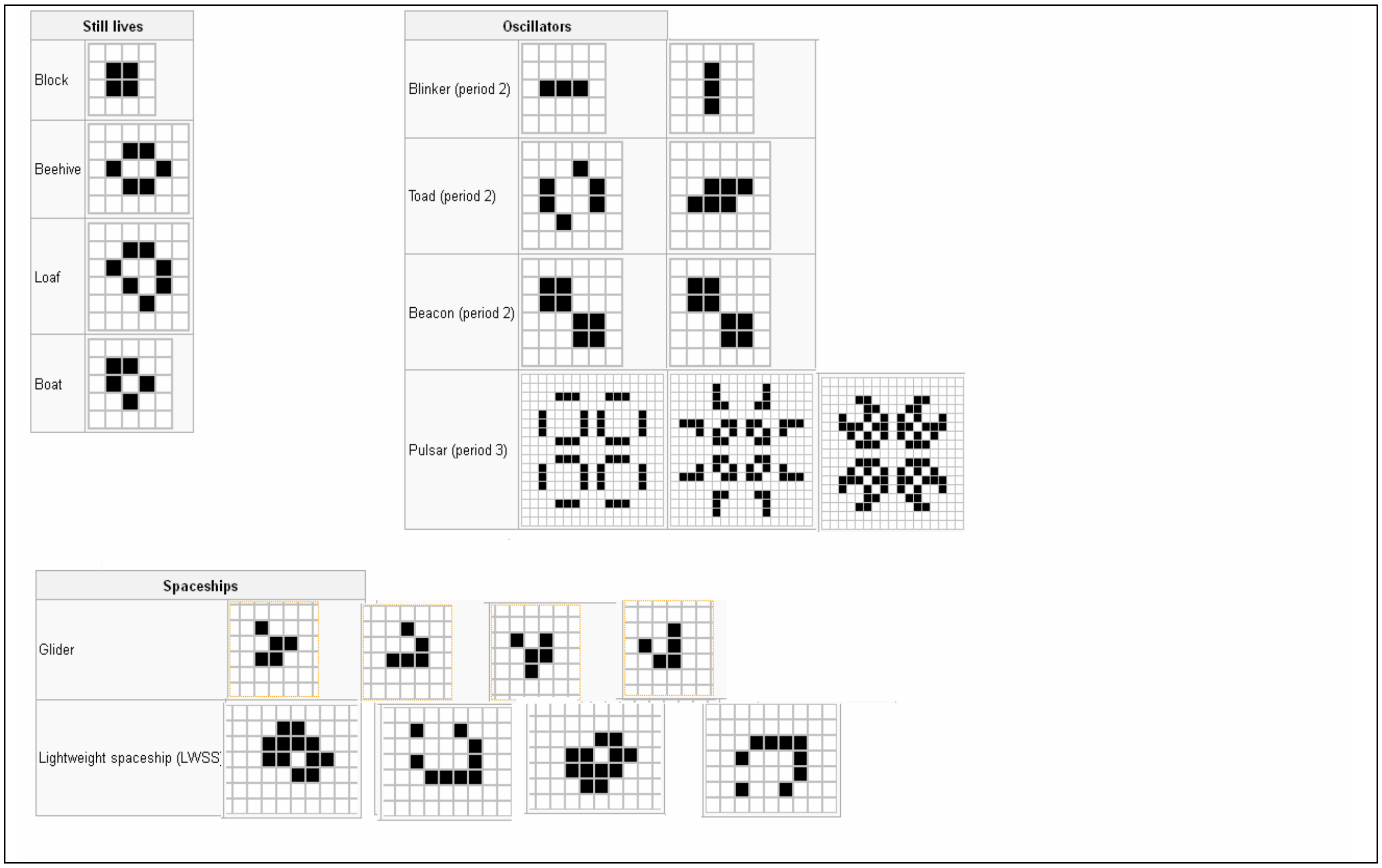
This particular cellular automaton has maximum number of two states, dead (0 or black) and alive (1 or white), From one tick of the clock to the next, the cells of the Game of Life cellular automaton can be either alive or dead, according to the following rules devised by Conway:

- Birth : If a cell is dead at time *t*, it comes alive at time *t*+1 if it has exactly 3 neighbors alive;
- Death : If a cell is alive at time *t*, it comes dead at time *t*+1 if it has :
  - (Lonliness) Fewer than 2 neighbors alive
  - (Overpopulation) More than 3 neighbors alive
  -



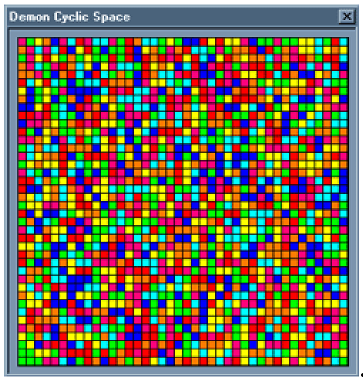
Game of Life Board

An amazing fact is that there are some shapes or patterns that repeat after a period of generations. The simplest static patterns are "Still Lives" that do not change over time. The repeating patterns named "Oscillators" (a superset of "Still Lives") do not move on the board but their shape repeatedly change during a certain generation period. The patterns that move across the board are called "Spaceships".



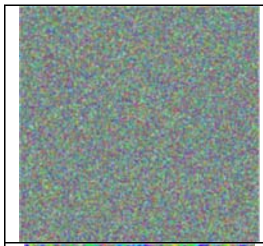
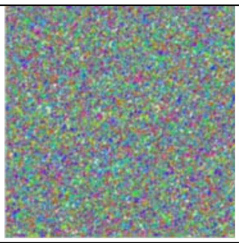
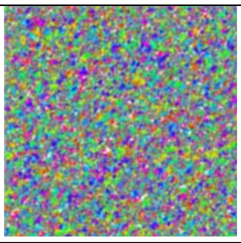
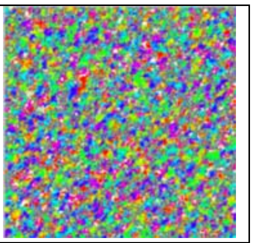
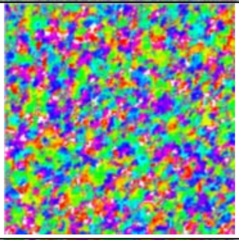
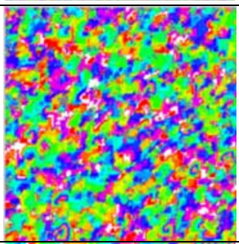
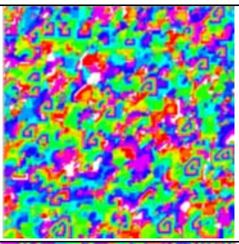
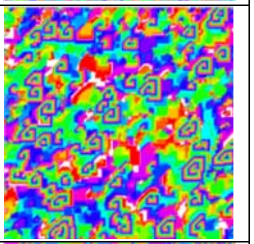
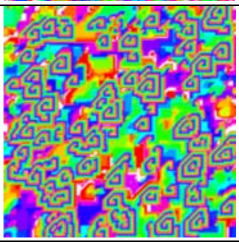
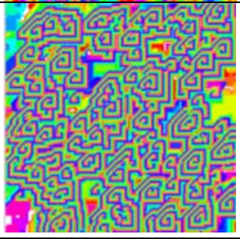
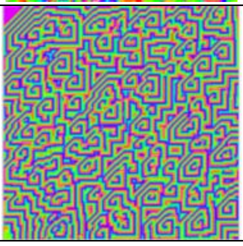
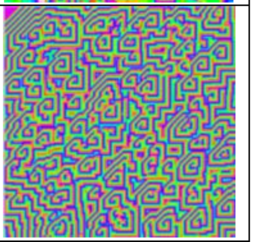
Demon Cyclic Space

This automaton has maximum number of n states. Each of the n possible states for a cell could be represented by a different color and numbered from 0 to *n*-1. A cell that happens to be in state *k* at one tick of the clock dominates any adjacent cells that are in state *k*-1 meaning that these adjacent cells change from *k*-1 to *k*. This rule resembles a natural chain in which a cell in state 2 can dominate a cell in state 1 even if the later is dominating a cell in state 0. But, since the automaton is cyclic, the chain has no end and a cell in state 0 dominates its neighboring cells that are in state *n*-1



Demon Cyclic Space Board

Initialized as a random distribution of colored cells, it always ends up with stable, angular spirals.

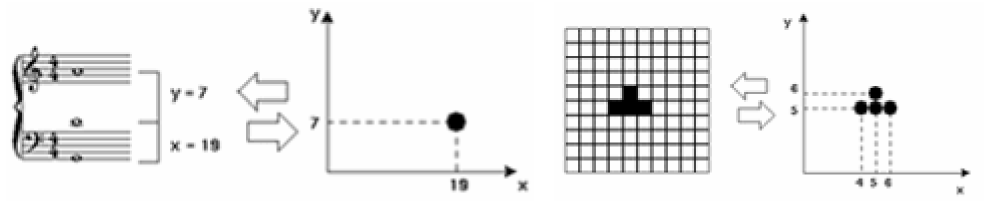
CA and Music

A CA is a pure mathematical structure that evolves over time. Visual representations of CA, as mentioned so far, demonstrate its time based character as a potential to be represented in music. Pictorial representations of some CAs can be quite striking and beautiful. Is it possible to come up with interesting and pleasing music using CAs and its evolution? Some composers like Xenakis[2] (Horos (1986)) or Synthesizers like Chaosynth[6], or algorithmic composition programs like WolframTones [13] or CAMUS [14] have used CA in their underlying structure. The followings explain some applied method to utilize CA in music composition or sound design.

Xenakis explains about CA in "Formilized Music" [15] : "The method (cellular automata) helps in deciding how to go from the notes of one chord to those of another within a rational, perceptible structure. (…) Let's say you have a grid on your screen, with vertical and horizontal lines forming small squares, that is, cells. There are empty. It's for the composer (whether working with pictures of with sounds) to fill them. How? One way is through probabilities, for instance by using the Poisson distribution, as I did 30 years ago in "Achorripsis". There's also another way, with the help of a rule that you work out for yourself. Let's suppose the vertical lines represent a chromatic scale, or semitones, quarter-tones and so on. Any kind. You start at a given moment, that is, at the given vertical line, at a given pitch – in other words, a cell – and you say: there's a note played by an assigned instrument. What's the next moment going to be? What notes? In accordance with your rule, the cell which has been filled gives birth to say, one or two adjacent cells. In the next step each cell will create one or two notes. Your rule helps to fill the entire grid. "

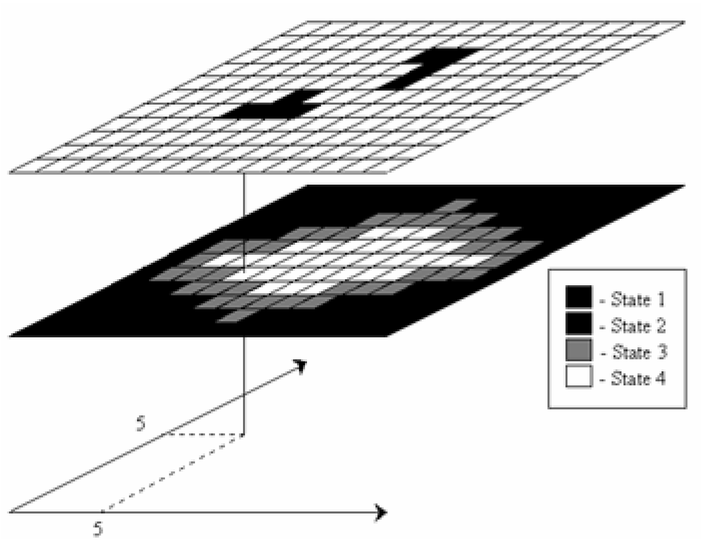
Acoording to WolframTones [13], a simple possibility to use CA in a piece, is to map each cell of a CA, e.g. Game of Life, to a specific pitch. When a cell is alive, the related pitch would be heard in each generation tick. A tempo could be used to advance each generation in time. In order to have multiple instruments, different CAs can be utilized, each of which responsible for a single instrument.

A more complex approach in CAMUS [14], is using a Cartesian model to represent a triple; that is, a set of three notes. The model has two dimensions, where the horizontal coordinate represents the first interval of the triple and the vertical coordinate represents its second interval, based on a predefined fundamental note.



The system uses both mentioned automata in parallel to produce music. The Game of Life automaton produces triples and the Demon Cyclic Space automaton determines the "Orchestration" of the composition. In this case, each color corresponds to an instrument designated to perform the notes generated by a specific cell. Each musical cell may have its own timing; also the notes within a cell can assume different durations and can be triggered at different times.

To begin the music process, the Game of Life automaton is set up with a starting configuration, the Demon Cyclic Space automaton is initialized with random states, and both are set to run. At each time step, the co-ordinates of each live cell are analyzed and used to determine a triple which will be played at the corresponding time in the composition. The state of the corresponding cell of the Demon Cyclic Space automaton is used to determine the orchestration of the piece. This configuration is illustrated below. In this case, the cell in the Game of Life at (5, 5) is alive, and thus constitutes a sonic event (that is, a set of three notes). The corresponding cell in the Demon Cyclic Space is in state 4, which means that the sonic event would be played by instrument number four (e.g., using MIDI channel 4). The co-ordinates (5, 5) describe the intervals in a triple: the fundamental pitch, the note five semitones above the fundamental, and the note ten semitones above the fundamental.



II. Csound Implementation

Generic CA

In order to model CA, we need a two dimensional matrix, or grid, having a size and maximum number of states and the current generation number. As there is no such a data structure in Csound, we have to implement 2D array using a one-dimensional table. So, there should be a mapping from 2D indexes (row, column) to linear table indexes. Table global variable and the mappings could be achieved by means of the following lines of code.

```
giSize init 5
giStates ftgen 1, 0, -giSize * giSize, -2, 0
giGeneration init 0
giMaxState init 15


opcode ToLinearIndex, i, ii
```

```
iRow, iColumn xin
   iLinearIndex init -1
   if iRow >= 0 && iRow < giSize && iColumn >= 0 && iColumn < giSize then
     iLinearIndex = iRow * giSize + iColumn
   endif
   xout iLinearIndex
endop


opcode To2DIndex, ii, i
iIndex xin
   iRow init -1
   iColumn init -1
   if iIndex >= 0 && iIndex < giSize * giSize then
      iRow = int(iIndex / giSize)
      iColumn = iIndex % giSize
   endif
   xout iRow, iColumn
endop
```

Based on the maximum number and states, the initial CA states, could be set by generating random numbers. The “`iCA`” parameter denotes CA, or function table in other words. In fact to generalize global variables for different CAs, we should use tables instead of single variables, each of which for an independent cellular automaton. So, we could have different CAs with different sizes or max states, etc. A better solution will be discussed in the object-oriented approach section. Until then, the simpler problem, with shared automata variables, would be dealt with.

```
opcode InitCA, 0, iii
iSize, iMaxState, iCA xin
   iIndex init 0
   giMaxState = iMaxState
   giSize = iSize
   initLoop:
      iValue random 0, iMaxState
      iValue = int(iValue)
      tableiw iValue, iIndex, iCA
   loop_lt iIndex, 1, giSize * giSize, initLoop
endop
```

A simple approach for showing the content of the table is using a Display Widget and sending the concatenation of the table states as a string for setting its text value. In case of using colors and being able to set the widget backgrounds, the user interface could be richer.

```
opcode RepaintCA, 0, i
iCA xin
   iIndex init 0
   iLastRow init 0
   SText = ""
   repaintCA:
      iState table iIndex, iCA
      iRow, iColumn To2DIndex iIndex
      SState sprintf "%d ", iState
      if iLastRow < iRow then ; instead of nested loops
         SText strcat SText, "\n"
```

```
            iLastRow = iRow
         endif
         SText strcat SText, SState
      loop_lt iIndex, 1, giSize * giSize, repaintCA

      outvalue "CAText", SText
   endop
```

Some utility opcodes for working with CA could also be implemented. GetCellState and SetCellState opcodes could be used for reading and writing the cell states in a specific row and column.

```
opcode GetCellState, i, iii
iRow, iColumn, iCA xin
   iIndex ToLinearIndex iRow, iColumn
   if iIndex < 0 || iIndex >= giSize * giSize then
      iCellState = 0
   else
      iCellState table iIndex, iCA
   endif
   xout iCellState
endop


opcode SetCellState, 0, iiii
   iCellState, iRow, iColumn, iCA xin
   iIndex ToLinearIndex iRow, iColumn
   tableiw iCellState, iIndex, iCA
endop
```

In order to count the number of neighbors of a specific cell, which are in a specific state the following opcodes could be implemented. The main opcode is `CountNeighboursInState` which count the number of neighbors of a cell in a row and column, which are in a certain state. There are eight neighbors for each cell and `AddCountIfInState` opcode is called for each of these neighbors respectively.

```
opcode AddCountIfInState, i, iiiii
iRow, iColumn, iState, iCount, iCA xin
   iNewCount = iCount
   iCellState GetCellState iRow, iColumn, iCA
   if iCellState == iState then ;IsAlive
      iNewCount = iCount + 1
   endif
   xout iNewCount
endop


opcode CountNeighboursInState, i, iiii
iRow, iColumn, iState, iCA xin
   iCount init 0

   iR = iRow - 1
   iC = iColumn - 1
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iC = iColumn
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iC = iColumn + 1
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iR = iRow
   iC = iColumn - 1
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iC = iColumn + 1
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iR = iRow + 1
   iC = iColumn - 1
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iC = iColumn
   iCount AddCountIfInState iR, iC, iState, iCount, iCA

   iC = iColumn + 1
   iCount AddCountIfInState iR, iC, iState, iCount, iCA
   xout iCount
endop
```

Game of Life CA

Game of Life is a CA that its maximum number of states is two. So its init opcode could call generic CA "`initCA`" opcode by passing '2' as maximum states:

```
opcode InitGameOfLifeCA ,0, ii
iSize, iCA xin
   InitCA iSize, 2, iCA
endop
```

The rule for Game of Life could be implemented by specifying the next value for each state based on the number of alive or dead neighbors.

```
opcode GetNextGameOfLifeState, i, iii
iRow, iColumn, iCA xin
   iCurrentState GetCellState iRow, iColumn, iCA
   iAliveNeighbours CountNeighboursInState iRow, iColumn, 1, iCA
   iNextState = iCurrentState
   if iCurrentState == 0 then ;if is dead
      if iAliveNeighbours == 3 then
         iNextState = 1 ;become alive
      endif
   endif
   if iCurrentState > 0 then; else (is alive)
      if iAliveNeighbours < 2 || iAliveNeighbours > 3 then
         iNextState = 0
      endif
   endif
xout iNextState
endop
```

By performing a loop over all the states in CA and calling `GetNextGameOfLifeState` opcode, the next generation of CA could be calculated. In order not to lose the values for CA states, a temporary CA (e.g. table function 999) is used to store the next states, and then copy the whole temporary table in the original CA.

```
opcode AdvanceGameOfLifeGeneration, 0, i
iCA xin
   iIndex init 0

   advance_gol_loop:
      iRow, iColumn To2DIndex iIndex
      iNextState GetNextGameOfLifeState iRow, iColumn, iCA
      ;999 is the default temp table
      SetCellState iNextState, iRow, iColumn, 999
    loop_lt iIndex, 1, giSize * giSize, advance_gol_loop

    iIndex2 init 0

    copy_gol_loop:
       iValue table iIndex2, 999
       tableiw iValue, iIndex2, iCA
    loop_lt iIndex2, 1, giSize * giSize, copy_gol_loop

    giGeneration = giGeneration + 1
endop
```

Demon Cyclic Space CA

A similar approach is taken for implementing “Demon Cyclic Space” CA and the corresponding opcodes are as the followings:

```
opcode InitDemonCyclicSpaceCA ,0, iii
   iSize, iMaxState, iCA xin
   InitCA iSize, iMaxState, iCA
endop

opcode GetNextDemonCyclicSpaceState, i, iii
iRow, iColumn, iCA xin
   iCurrentState GetCellState iRow, iColumn, iCA
   iNextState = iCurrentState
   if iCurrentState == giMaxState then
      iDominateState = 0
   else
      iDominateState = iCurrentState + 1
   endif
   iDominateNeighboursCount CountNeighboursInState iRow, iColumn, 
iDominateState, iCA
   if (iDominateNeighboursCount > 0) then
      iNextState = iDominateState
   endif

   xout iNextState
endop
```

```
opcode AdvanceDemonCyclicSpaceGeneration, 0, i
iCA xin
   iIndex init 0
   advance_dcs_loop:
      iRow, iColumn To2DIndex iIndex
      iNextState GetNextDemonCyclicSpaceState iRow, iColumn, iCA
      ;999 is the default temp table
      SetCellState iNextState, iRow, iColumn, 999
   loop_lt iIndex, 1, giSize * giSize, advance_dcs_loop

   iIndex2 init 0

   copy_dcs_loop:
      iValue table iIndex2, 999
      tableiw iValue, iIndex2, iCA
   loop_lt iIndex2, 1, giSize * giSize, copy_dcs_loop

   giGeneration = giGeneration + 1
endop
```

In fact, instead of having two different opcodes for advancing generation, only a general opcode would work for both of cellular automata types. The only difference is calling the opcode for NextState. This would be detailed more in the object-oriented approach section.

Instruments and Score

A simple instrument and score could be something like the following. For example based on a parameter, e.g. p2, an initialization or advancing event could be sent to an instrument, with and underlying CA present.

```
instr 1
   if p2 == 0 then
      InitDemonCyclicSpaceCA 5, 10, 1
      ;InitGameOfLifeCA 1
      RepaintCA 1
   else
         AdvanceDemonCyclicSpaceGeneration 1
         ;AdvanceGameOfLifeGeneration 1
      RepaintCA 1
   endif
endin
</CsInstruments>

<CsScore>
i 1 0 2
i 1 3 2
i 1 6 2
i 1 9 2
</CsScore>
```

III. Object-Oriented Approach

In this section, with a very brief explanation of object-oriented concepts, the same problem would be designed considering this point of view.

Class: Member variables + Methods

Using object-oriented concepts, instead of having data structures and opcodes as separate elements, a so called “Class” integrate these two as a whole. So a “Class” contains two parts: attributes or member variables, and methods or operations. The former represenst data while the latter represents the behavior. This idea somehow is close to the model of real world. For instance, there is an Animal class having some attributes, like age, gender, size, etc and also some behaviors like walking, eating, sleeping. So, the so-called UML “Class Diagram”[12] of Animal is like this:

Animal
age gender size
Walk Eat Sleep

The first row is the name of the class, the second one the attributes and the last one the methods.

In other words, a “Class” integrates data and algorithm together. So, its behavior can use and change its attributes. Therefore, considering Csound, global variables could be member variables and opcodes could be methods of a class. The benefits of this way of thought would be more clarified in the succeeding sections.

In our CA example, iStates, iSize, iMaxState and iGeneration are proper samples of attributes of CellularAutomata class. As well as oce could think of size or gender is an attribute of an animal, we would mention size or maximum states of a CA. These are the properties of CA which the values represent the “Instance” of the class. There could be different instances of a class with different attribute values.

In most of the opcodes, the last `iCA` parameter is used to specify which CA the related opcode is called for. By means of having a class, all of the `iCA` parameter will be omitted and opcodes will be simpler.

Visibility

Member variables and methods of a class could have different kinds of visibility. To explain this concept, we could take a car as an example. The engine of car is not visible to a driver, while the steering wheel is visible and is the interface between car and the driver. We could name engine as a “private” and steering wheel as a “public” part of a car.

As an example, `AddCountIfInState` opcode is a proper "private method" candidate for CA, while `AdvanceGeneration` is a suitable "public method" which could be called by a developer as its user. The first opcode actually is a helper method for the other ones and does not seem meaningful to be called by a developer. The same story could be valid for attributes. For reaching a higher level of so-called "Encapsulation" and security, one can define the entire member variables as private, only letting them change by means of calling their corresponding "setter" methods.

Inheritance

There is a similarity between the relation between Animal and Elephant, and Cellular Automata and Game of Life. In the real world, there is a class hierarchy between the classes. For instance, Elephant is a sub-classes Animal, which in turn has different sub-classes like AfricanElephant and IndianElephant. So, there is a hierarchy or tree like structure, Animal as the top node, Elephant below it, and AfricanElephant and IndianElephant as the most bottom level. Each level has the same characteristics of the upper level, while it is more specialized regarding some aspects. This concept is exactly the same in CA with two sub-classes, GameOfLifeCA and DemonCyclicSpaceCA. The phrase : "GameOfLifeCA is a CA with MaxStates as two", suggests this kind of relationship. Generally, "is a" can be regarded as this kind of sub-class relationship or "Inheritance". The sub-class inherits the member variables (not private ones) of the super-class and adds some other new ones.

Another visibility level which we could think of, is being "protected". In addition to "public" and "private" visibility, a "protected" member variable or method, is not able to be called outside a class, but could be used and overridden by the sub-classes.

Method Overriding

When a sub-class extends a super-class by means of "Inheritance", it not only can add some attributes or behaviors to it, but also change some behaviors and define its own. For example although there is a default implementation for Elephant::Walk, AfricanElephant::Walk can replace this default, "Overriding" the default implementation. In CA example, The default implementation for `GetMaxState` is returning the `iMaxState` member variable. But `GameOfLifeCA` overrides this method, always returning '2'.

There could be a possibility for a sub-class to call the super-class implementation. For instance, when a sub-class method adds some behavior to the overridden super-class method, it could be able to call the super-class method first, and then add some specific logic,

Abstract or concrete

The inheritance relationship has another innate aspect, related with "Method Overriding": "Being abstract or concrete". A class is "Abstract" if it has some abstract methods resulting that this class can not have any instances. For example, Animal class is an abstract class, because it has some abstract methods. We can not exactly describe how an animal walks in general. It is so abstract and only in its sub-classes this behavior could be defined in details. In other terms, we can not have an instance of an general Animal, it should be either an elephant, a lion, a cat, etc. So the Animal class is abstract and its sub-classes define the abstract behaviors and become concrete. In our example, CA is exactly the same. As long as no rule for CA in general is defined, this class remains abstract. Therefore, the method related to this rule should be an abstract method and the sub-classes should “Override” it to define the behavior and declare some concrete classes such as `GameOfLifeCA`.

Template Method, Hook Method

There are some methods which their implementations are fixed, but the methods which are called by this method may change in the sub-classes. The former type of methods are called “Template Method” [10] while the latter is called “Hook Method”. For instance, `AdvanceGeneration` method is always the same for all the CA sub-classes. It loops over the states calls `GetNextState` method for each state. But `GetNextState` method defines the rule of game and is an abstact method in CA base class. So, in the sub-classes there is no need to re-implement `AdvanceGeneration` method, and only `GetNextState` should be overridden. The first method acts as a template, having a hook, acting as an empty slot which should be filled. This helps to reduce the amount of implementation required for a class library or its usage. Using this pattern, a big template method may call only one hook method, and by overriding only the hook method the logic may change in the sub-classes of a base class. The `GetMaxState` method is another proper example of hook method.

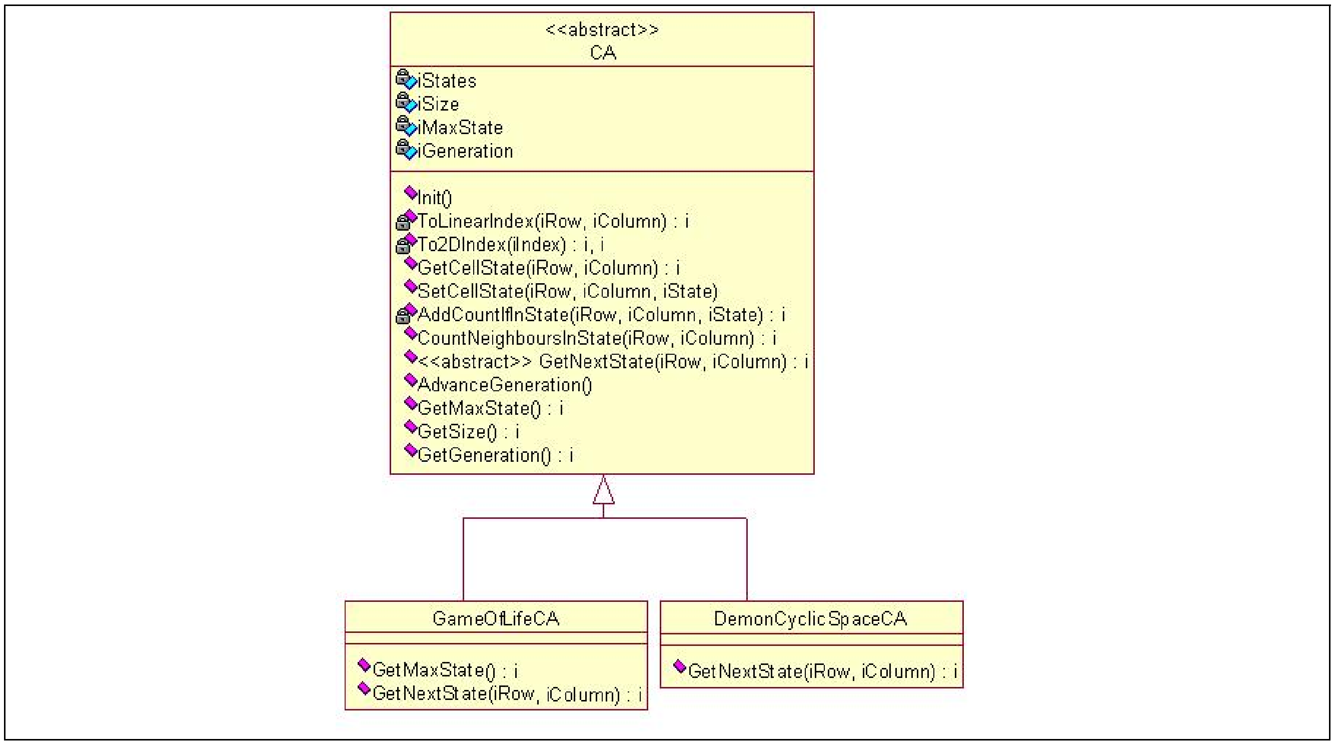
Instantiation

Instantiation is the process of taking an instance of a class. When referring to a cat in general, one can not talk about its age. But a specific instance of “Cat” class has some specified attribute values, including age. When instantiating a class, an “Instance” or “Object” of that class gets created , and its attributes get concrete values. A “Constructor” is a method of the class which gets called in the process of instantiation. So, the initialization logic of a class could be defined in its constructor. Each class can have different constructors with different number of parameters. For example, a CA can have a constructor with a MaxStates parameter, and another parameter with two MaxStates and Size parameters. The name of constructor method could be a keyword like “Init” (In many object-oriented programming languages, name of the constructor is the name of the class itself). In other words, using the “Init” keyword as a method name, changes its type to a “Constructor”. When instantiating a class, the constructor will be called automatically

So, an object is considered as a specific variable and the Csound parser should know its class as a new variable types. The class member variables may be of different rates, e.g. i-rate or k-rat or a-rate, just like normal variables for each opcode.

Class Diagram and the corresponding code

All in all, the object-oriented facilities and patterns could lead to having three classes in CA example denoted by the following diagram. As shown, the two sub-classes only need to override `GetNextState` method to define the rule of the game. `GameOfLifeCA` also overrides `GetMaxState` to return the fix value of ‘2’ for the maximum states. The icon to the left of the attributes or behaviors UML class diagram[12] depicts the visibility, as mentioned so far.



<<abstract>>
CA
iStates
iSize
iMaxState
iGeneration
Init()
ToLinearIndex(iRow, iColumn) : i
To2DIndex(iIndex) : i, i
GetCellState(iRow, iColumn) : i
SetCellState(iRow, iColumn, iState)
AddCountIfInState(iRow, iColumn, iState) : i
CountNeighboursInState(iRow, iColumn) : i
<<abstract>> GetNextState(iRow, iColumn) : i
AdvanceGeneration()
GetMaxState() : i
GetSize() : i
GetGeneration() : i
GameOfLifeCA
GetMaxState() : i
GetNextState(iRow, iColumn) : i
DemonCyclicSpaceCA
GetNextState(iRow, iColumn) : i

The corresponding Csound code could be something like the following. The suggested added keywords to Csound, thinking of Csound++, are marked as bold. All the “iCA” parameters have been eliminated because of the integration of opcdoes inside the class. The opcode names have also been simplified in the classes, because of the mentioned feature.

```
class CA
   private iSize init 5
   private iStates ftgen 1, 0, -giSize * giSize, -2, 0
   private iMaxState init 15

   public opcode GetMaxState i, 0
      xout iMaxState
   endop

   public opcode GetSize i, 0
      xout iSize
   endop

   private opcode ToLinearIndex, i, ii
   …
   private opcode To2DIndex, ii, i
   …
   public opcode Init, 0, ii
   …
   public opcode Repaint, 0, 0
   …
   public opcode GetCellState, i, ii
   …
   public opcode SetCellState, 0, iii
   …
   private opcode AddCountIfInState, i, iiii
   …
   protected opcode CountNeighboursInState, i, iii
   …
   protected abstract opcode GetNextState, i, ii

   public opcode AdvanceGeneration, 0, i
   …
endclass



class GameOfLifeCA inherits CA
   public opcode GetMaxState, i, 0
   protected abstract opcode GetNextState, i, iii
   …
endclass

class DemonCyclicSpaceCA inherits CA
   protected abstract opcode GetNextState, i, iii
   …
Endclass

instr 1
   if p2 == 0 then
      DemonCyclicSpaceCA ADemonCyclicSpaceCA 5, 10, 1 ; Instantiation
      ADemonCyclicSpaceCA.RepaintCA 1 ; Calling method of an instance

      …
   endif
endin
```

Conclusion

Having the benefits of an object-oriented language or environment in Csound, like SuperCollider or OpenMusic, does not imply that the developer has no other choice using it. If a developer desires, he or she could continue implementing without using object-oriented facilities. But it really helps to model the problem more easily leading to provide a more powerful class library. Moreover, in case of lack of the object-oriented programming features, one could go on using object-oriented design patterns and applying them in a non-object-oriented programming language.

There could be more object-oriented features integrated into Csound in addition to having options like : classes to integrate opcodes and member variables, visibility, inheritance, method overriding, abstract classes and methods, template and hook method. But any of these features, for example opcodes visibility, could help having a more strong-typed language

Acknowledgements

References and links

1- Csound FLOSS Manual http://www.flossmanuals.net/csound/
2- Object-oriented Design http://en.wikipedia.org/wiki/Object-oriented_design
3- Object-oriented Programming http://en.wikipedia.org/wiki/Object-oriented_programming
4- Makis Solomos, Cellular Automata in Xenakis' Music. Theory and Practice. Defnitive Proceedings of the Inter-national Symposium Iannis Xenakis, 2005.
5- Introduction to Cellular Automata Music Research, Eduardo Reck Miranda, http://x2.i-dat.org/~csem/UNESCO/8/8.pdf
6- Eduardo Reck Miranda. Evolving Cellular Automata Music: From Sound Synthesis to Composition. Proceedings of the Workshop on Arti_cial Life Models for Musical Applications, 2001.
7- Gardner, M. (1970). “Mathematical Games: The fantastic combinations of John Conway’s new solitaire game ‘life’”, Scientific American, 223(10), pp 120-123.
8- Millen, D. (1990) “Cellular Automata Music.” In S. Arnold & D. Hair, Eds., *Proceedings of the 1990 InternationalComputer Music Conference*, pp. 314-316. San Francisco: ICMA.
9- Peter Beyls. The Musical Universe of Cellular Automata. ICMC Proceedings, pages, 1989 James McCartney.
10- Rethinking the Computer Music Language: SuperCollider. Computer Music Journal, 2002.
11- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson , John M. Vlissides
12- Object Management Group, http://www.uml.org/
13- How WolframTones work http://tones.wolfra.com/about/how.html
14- Introduction to Cellular Automata Music Research http://x.i-dat.org/~csem/UNESCO/8/index.html
15- Xenakis, I. (1992). Formalized Music, English translation C. Butchers, G. H. Hopkins, J. Challifour, second edition with additional material compiled, translated and edited by S. Kanach, Stuyvesant (New York), Pendragon Press.