Scintillae: How to Approach Computing Systems by Means of Cellular Automata

Gabriele Di Stefano[1] and Alfredo Navarra[2]

[1] Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica,
Università degli Studi dell'Aquila, Italy
gabriele.distefano@univaq.it
[2] Dipartimento di Matematica e Informatica,
Università degli Studi di Perugia, Italy
alfredo.navarra@unipg.it

Abstract. The paper deals with a very simple game called *Scintillae*. Like in a domino game, Scintillae provides the player with limited basic pieces that can be placed over a chessboard-like area. After the placement, the game starts in a sort of runtime mode, and the player enjoys his creation. The evolution of the system is based on few basic rules.

Despite its simplicity, Scintillae turns out to provide the player with a powerful mean able to achieve high computational power, storage capabilities and many other peculiarities based on the ability of the player to suitably dispose the pieces.

We show some of the potentials of this simple game by providing basic configurations that can be used as "sub-programs" for composing bigger systems. Moreover, the interest in Scintillae also resides in its potentials for educational purposes, as many basic concepts related to the computer science architecture can be approached with fun by means of this game.

1 Introduction

Scintillae is a game designed for fun to simulate a sort of domino effect on a PC.[1] Although it hides some peculiarities that cannot be realized by standard domino games with falling pieces, it is very interesting how such kind of simulator may easily realize a computing system in a sort of visual programming environment [3]. Its strength is witnessed by the high computational power that indeed can be obtained by means of the few pieces provided by the game for composing desired configurations. Instead of domino pieces falling in a sequence determined by their proximity, Scintillae provides the user with sparks that seem to move according to designed paths obtained by disposing arrows on a chessboard-like area. The rules that establish the evolving of the system are very simple.

1.1 Rules of Scintillae

In an unbounded area divided into squares, it is possible to place two basic pieces, at most one per square, to compose a game: *Sparks* ("Scintillae" in the

[1] An executable graphic version of the program along with some explanatory examples contained also in this paper can be found in [1,2].

G.C. Sirakoulis and S. Bandini (Eds.): ACRI 2012, LNCS 7495, pp. 534–543, 2012.

Latin language), and four types of *Arrows* ($\rightarrow$, $\uparrow$, $\downarrow$, $\leftarrow$), see Fig. 1. Each arrow has four neighboring squares. The square pointed by the arrow is called the *output square* while the other three squares are called *input squares*. The system is synchronous and pieces interact at each time t according to the next simple operations applied sequentially, that move the system's state to time $t + 1$:

1. an arrow becomes *loaded* if among its input squares there is exactly one spark;
2. each spark disappears and leaves empty the corresponding square;
3. for each loaded arrow, a spark is placed in the output square, if empty;
4. each loaded arrow becomes unloaded.

Fig. 1 shows the available pieces and some basic configurations. By the rules above, it follows that arrows never change from their original placement. That's all! The rules are specified, now only the imagination of the user can lead to surprising results. In the implemented program, once the pieces have been placed, a running button can be pressed to enjoy the evolution. Another way to experience the evolution of the system is to press the space bar for a single clock's tick.

Clearly, Scintillae belongs to the family of *Cellular Automata* [4,5,6,7,8,9]. Similarly to such systems, in Scintillae the user is provided with an area divided into cells. Each cell may change its status according to its own one, and to the status of a limited number of neighbors. The evolution is synchronous for all the cells. Hence, Scintillae preserves the fundamental peculiarities of such kind of computational means, that are: parallelism, locality, and homogeneity. In fact, all the cells are updated synchronously in parallel, by applying the same common rule based on local peculiarities. Likewise domino games but differently from most cellular automata, in Scintillae there are predefined paths provided by the initial disposal of the arrows that determine the evolution of the sparks. It reminds the WireWorld tool [9], but it is even simpler as less rules are specified. Moreover, the composed configurations appear very neat, clear and intuitive, hence more appropriate for educational purposes. However, "unexpected" but useful behaviors may occur for particular configurations.

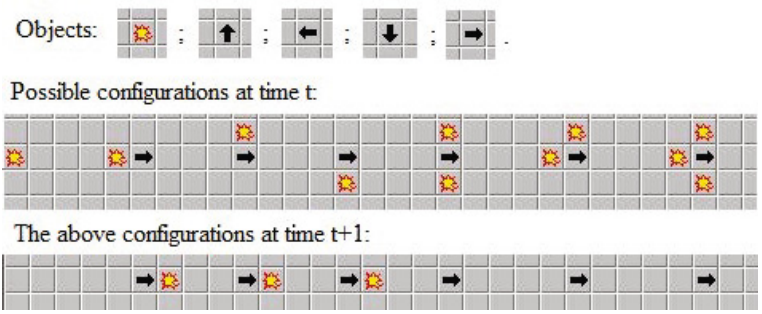


Fig. 1. Available pieces, and a description of the evolving synchronous system

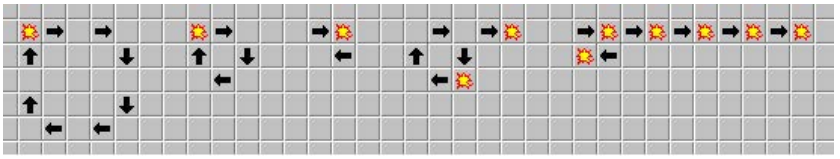


Fig. 2. Three possible cycles with periods 8, 4, and 2, respectively, and their application in generating infinite sparks

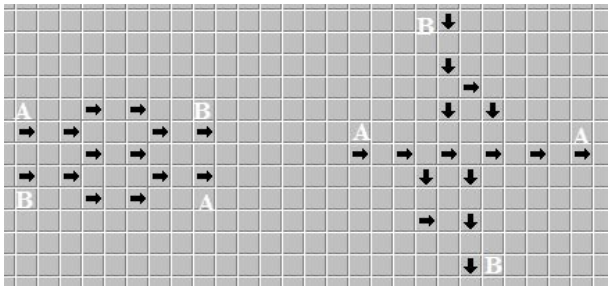


Fig. 3. Two possible ways of realizing crossings

2 Basic Configurations

One main peculiarity of Scintillae is the opportunity to obtain infinite evolutions. In fact, if the user defines a cycling sequence of arrows, and place a spark nearby one of them, the spark will be moved along the cycles infinitely often, until the user stops the run-time mode. In Fig. 2, three possible cycles are shown. Moreover, on the right, it is shown how cycles can be used as infinite generators of sparks. In particular, the size of the used cycle determines the frequency of sparks in output. Another way to decide the frequency of sparks is to insert more sparks in the cycle at the beginning during the designing mode.

Interesting basic configurations concern the appropriate placement of the pieces in order to realize desired connections. In particular, it may happen that the user requires two sequences of arrows (*lines*) to cross each other in order to move sparks towards desired destinations. The problem that arises is that sparks on a line could be duplicated on the other line at the crossing point. Surprisingly, Scintillae provide a way to realize crossings. As shown in Fig. 3, two sequences of arrows may cross in two different ways, horizontally or vertically, and it is interesting to observe during the run-time mode how the sparks follow the appropriate A or B paths, without interfering.

A useful configuration that recall a candle or an on/off switch is shown in Fig. 4. If a spark is placed nearby the leftmost arrow →, the system moves to the rightmost configuration in few clock's ticks. The configuration remains the same until another spark enters the system. If this happens, again the first configuration occurs (switching off the candle). The rational behind such a structure

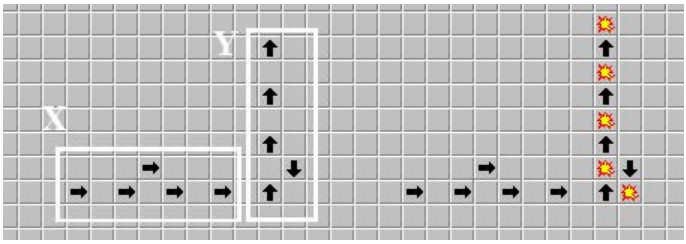


Fig. 4. On the left, the starting configuration representing a candle (switch) divided into two sub-structures X and Y. On the right, the configuration obtained after placing a spark near by the leftmost arrow.

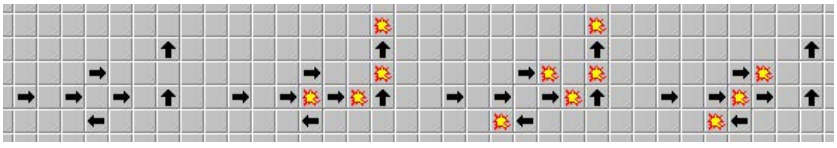


Fig. 5. The first configuration represents an alternative way for realizing a switch. The second and the third configurations are obtained alternately once a spark is placed near by the leftmost arrow. The last configuration can be obtained when switching off the candle, according to the time when a second spark arrives.

is provided by the two sub-structure depicted in the figure. First, an input spark is duplicated by means of the sub-structure X. Then, the two consecutive sparks arrive to the cycle in the sub-structure Y. If the candle is off (i.e., the cycle is empty), then the effect of the two sparks is that of filling the cycle, and hence the candle turns on. If the candle was on, then the two consecutive sparks remove the ones contained in the cycle since the entering arrow of the cycle will be neighbor of two sparks for two steps. Another way for realizing the candle is shown in Fig. 5. However, this configuration reveals a weird side effect. In fact, if a spark is placed near by the leftmost arrow, the system moves to an instable situation where the candle is switched on, but the configuration alternates between the second and the third ones shown in the figure. When another spark enters the system, the candle is switched off, but according to the time this new spark arrives, the configuration might become either the original one or the fourth one. Clearly, the configurations shown so far represent only few samples with respect to the potentials of Scintillae. The ability of the user to find new ways of composing the available pieces might realize surprising configurations.

3 From the Game to Computing

In this section, we exploit some of the potentials of Scintillae in order to realize computing systems. An interesting peculiarity of Scintillae is its nice attitude to provide a way for realizing combinatorial circuits. We now show how to implement the basic logic gates like *XOR*, *OR*, *NOT* and *AND*.

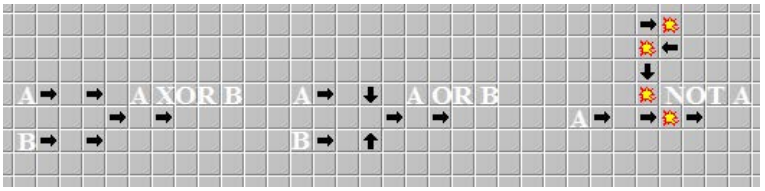


Fig. 6. *XOR*, *OR*, and *NOT* realized on Scintillae

The most natural logic gate obtainable by composing few lines of arrows is the *XOR*. As shown in Fig. 6, it is enough to put a line of arrows between the two lines carrying the two inputs of the gate. In fact, the first arrow of the output line will move a spark towards the output only if one single spark arrives from the input lines. When both the input lines carry on a spark, then the first arrow of the output line will be neighboring two sparks that disappear at the next clock's tick, as shown in the fifth configuration of Fig. 1.

Remark 1. Actually, a single arrow in Scintillae represents a *XOR* gate of three inputs, the neighboring squares not pointed by the arrow. It follows that any other logic gate obtainable by composing Scintillae's pieces is the result of composing *XOR* gates. Although the *XOR* is not a universal gate like the *NOR* and the *NAND* gates (see [10]), i.e., not all the other logic gates can be obtained from the *XOR*, we show how this problem can be overcome in Scintillae. Even tough this seems a contradiction, it will be better clarified later on.

Realizing the *OR* gate is also quite easy since it requires a little modification with respect to the *XOR*, as shown in Fig. 6. Now, if both the input lines carry on a spark, they will be both moved to the tail of the output line, where it appears like only one spark as output. In order to realize the *NOT*, a bit more understanding is required. In fact, we need to have a spark in the output line when there is nothing in input. In order to realize such a configuration, we make use of an infinite generator of sparks, as shown in the fifth configuration of Fig. 2. As shown in Fig. 6, when the input line carries on a spark, the middle arrow of that line will be neighboring with two sparks, as in the sixth configuration of Fig. 1, and then there won't be a spark at its head in the next clock's tick.

Remark 2. In the construction of the *NOT* gate resides the trick to obtain any other logic gate by means of *XOR* gates. In fact, once we have both the *OR* and the *NOT* gates we can obtain any other gate. The fact is that we are using also sparks to realize the *NOT* gate. In particular, we are able to generate a sequence of infinite sparks representing a line set to 1, and this is not possible when considering only *XOR* gates.

Concerning the *AND* gate, by simply applying De Morgan's rule [10], it can be realized as $NOT(NOT(\mathrm{A})\ OR\ NOT(\mathrm{B}))$. This is shown in the first configuration of Fig. 7. In the same figure, the other configuration still realizes the *AND* gate

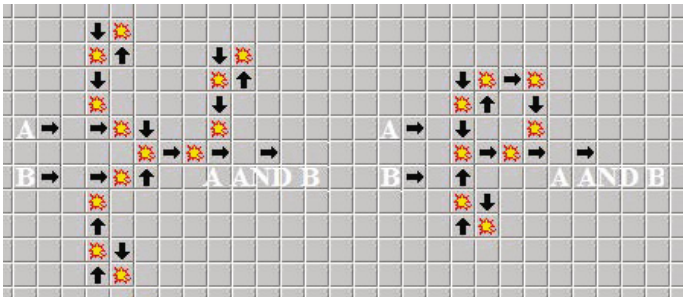


Fig. 7. Two configurations realizing the *AND* gate

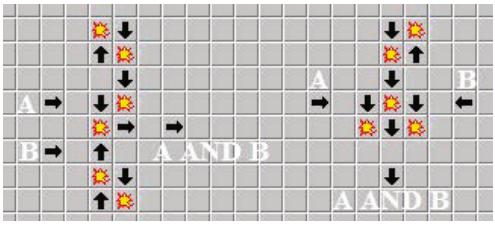


Fig. 8. Other two possible ways for realizing the *AND* gate

but some more insights are required for its understanding. It comes from the requirement of obtaining the "easiest" way for realizing such a gate. As part of the game, we tried to obtain the same results by means of optimized configurations in terms of used pieces. For the case of the *AND* gate, for instance, while the first configuration of Fig. 7 is quite straightforward since obtained by applying well-known rules, the other comes from our intuitions. Indeed, the rational behind it is simply to merge the infinite generators of the first configuration as much as possible. More tricky configurations that realize the *AND* gate are shown in Fig. 8. These required much more confidence with the game.

Another interesting circuit obtained by serializing some candles and appropriately connecting them with infinite generators, realizes a binary counter as shown in Fig. 9. The assumption is that 1 corresponds to the candle switched on, 0 otherwise. In the figure, it is shown a snapshot of the system after injecting eleven sparks as input (rightmost arrow ←).

In Fig. 10, we show a memory component of one byte with recorded value 11011100. The byte is contained in the cycle M. In order to read the value contained in the memory, one has to insert a spark in the read line R. For a correct reading of the byte, the input spark must arrive at line R synchronously with a spark that in the CLOCK cycle occupies the bottom leftmost square as in the figure.[2] Then a copy of the correct sequence describing the byte will flow

[2] The cycle defining the CLOCK is not really part of the memory, but is visualized only for a correct use of the memory component.

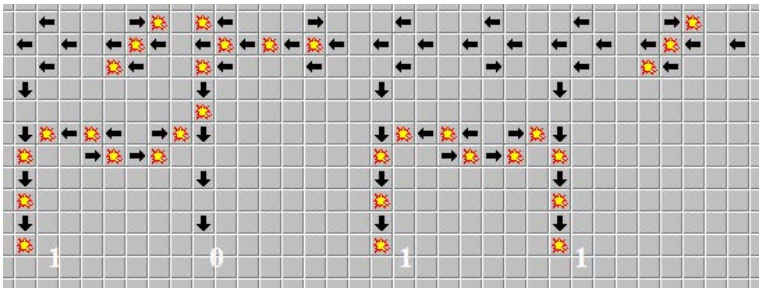


Fig. 9. A binary 4-bits counter

on the output line O. This is realized by temporarily open the "tap" $T3$ between M and line O. A tap like $T3$ is composed of a candle and two paths that leads to the same arrow belonging to a path. A closed tap is realized by switching on the candle and hence making two sparks neighboring to the common arrow of a line, i.e., blocking the flow on that line. To open the tap, it is sufficient to switch off the candle, hence removing the block. Note that, on the way from R to $T3$, the line $P2$ is used to duplicate the input spark in such a way that, after eight clock's ticks, the tap is automatically closed, hence letting flow only eight bits on the output line O. The input line C is used to clean the memory. In fact, when a spark enters this way, it is duplicated by means of line $P1$ in such a way that the two sparks reach the tap $T1$ and make it closed for exactly eight clock's ticks. In doing so, two sparks become neighbors to a common arrow of M, hence obtaining the delation of the contained byte. By using the C line, one obtains also another effect, that is, to open tap $T2$ for exactly eight clock's ticks. This is, in fact, used for writing in the memory. To write a new byte in the memory, the sequence of sparks describing the new byte must arrive at the input line W concurrently with a spark at line C. Hence, M is first cleaned and then refilled with the new sequence. For the synchronization of such operations we were required to carefully consider the length of the involved lines.

4 A Case Study

Based on some previous configurations, we might be able to simulate any combinatorial circuit. In this section, we aim to construct a circuit that counts from 0 to 3, cyclically, and displays the outcome on a standard seven-bars display. Moreover, we make also the display by means of the Scintillae's pieces. In doing so, we provide the evidence that Scintillae can be used for both computing and wider purposes more related to aesthetical factors. In Fig. 11, the mentioned configuration is shown. The snapshot is taken while the counter is displaying the number 2 on the seven-bars display realized by means of arrows. We are now going to describe all the "objects" composing the whole circuits. In order to realize the desired counter we need four main sub-circuits: (i) a clock that

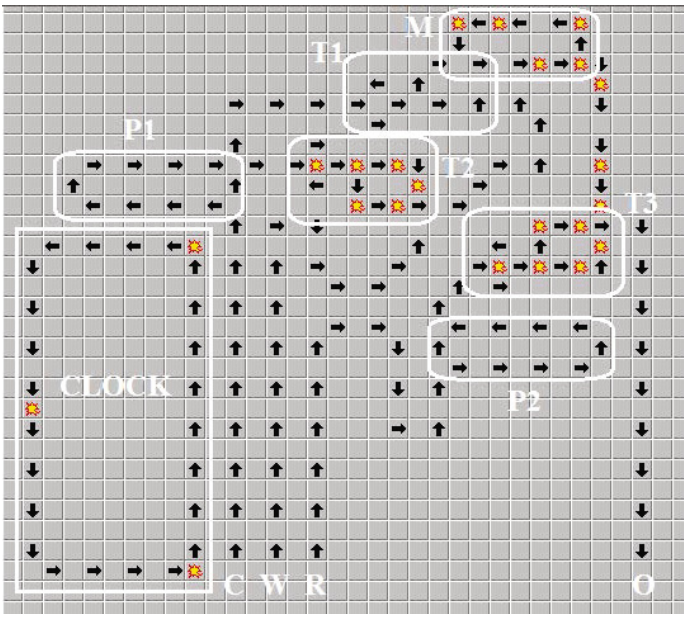


Fig. 10. A byte memory cell

frequently generates a signal in order to advance the counting; (ii) a counter that translates the received signal from the clock into a binary string representing how many signals have been received modulo the size of the counter (four in our case); (iii) a seven-bars display that has seven input lines for switching on the correspondent bars (see Fig. 12); (iv) a circuit able to convert the binary string representing a digit among the set $\{0, 1, 2, 3\}$ into the suitable signals that switch on the appropriate bars on the display for a correct visualization.

As discussed in the previous section, (i) and (ii) have been already realized. In fact, (i) is obtained by a cycle of arrows with one spark inside (see bottom left side of Fig. 11). In this way, we generate a spark every time the one in the cycle covers all the cycle. Clearly, the more is the length of the cycle, the less is the speed of the counter. The output of such a cycle is connected to a 2-bits counter (similar to the one shown in Fig. 9, see on the left side of Fig. 11) that provides the correct binary coding of the number of sparks received in input, modulo four. The construction realizing (iii) can be easily recognized on the right part of Fig. 11, with its seven input lines surrounding the display. This has been realized by means of a suitable disposal of the arrows, some of which are there only for aesthetical reasons. Concerning (iv), the circuit for converting the binary string into switching signals for the seven-bars display is shown in Fig. 12, along with the activation functions related to the bars of the display. Such a circuit is a bit hidden in Fig. 9, but following the lines from the counter to the display, one can easily recognize the logic gates used. Indeed, many of the arrows are used for crossing lines by means of the horizontal configuration shown

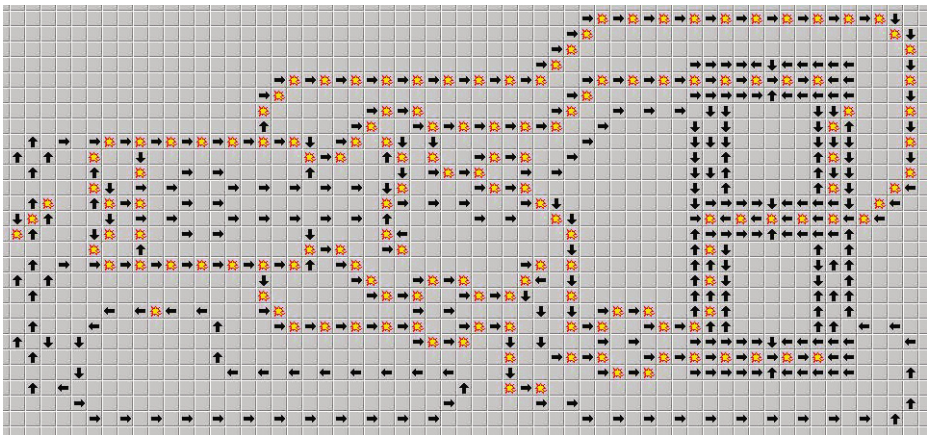
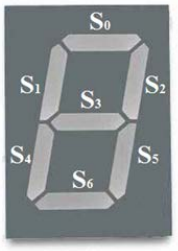


Fig. 11. Snapshot of a cyclic counter form 0 to 3 while displaying number 2



X	Y	S0	S1	S2	S3	S4	S5	S6
0	0	1	1	1	0	1	1	1
0	1	0	0	1	0	0	1	0
1	0	1	0	1	1	1	0	1
1	1	1	0	1	1	0	1	1

S0 = S6 = X OR NOT(Y);
S1 = NOT(X) AND NOT(Y); S2 = 1; S3 = X;
S4 = NOT(Y); S5 = NOT(X) OR NOT(Y)

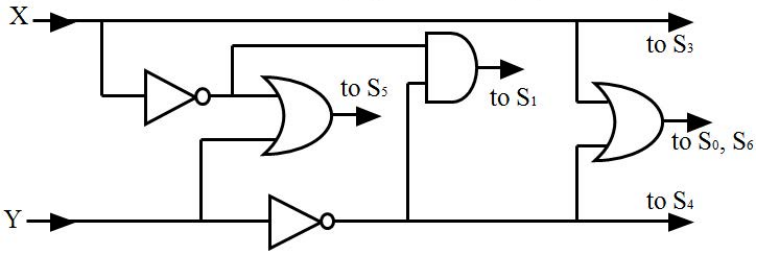


Fig. 12. The seven-bars display, the activation functions of its input lines, and the circuit used to convert a binary string of two bits XY into the appropriate signals for visualizing the input number by means of a decimal digit on a seven-bars display

in Fig. 3. Concerning, for instance the activation function of S_1, the *AND* gate has been realized by means of the first configuration shown in Fig. 8. Concerning S_2, since the specific circuit counts cyclically from 0 to 3, its activation function is always set to 1. This has been realized by embedding directly in the display an infinite generator of sparks. Moreover, it is interesting to note that, if we

add more consecutive sparks in the cycle that provides the input to the counter, we may obtain the visualization of the digits corresponding to the current digit plus the number of sparks in the cycle modulo four. In particular, if we put three consecutive sparks in the cycle, we obtain a cyclic countdown from 3 to 0. Actually, the same effect might be realized by reducing the length of the cycle so much that the display is not able to visualize all the input sequence. This implies a refresh frequency of the designed display that must be carefully managed by prolonging or shortening the connecting lines.

5 Conclusion

We have presented Scintillae, a new and simple cellular automaton that reveals high computational power capabilities. Surprising results have been obtained by suitably placing the few pieces provided by the game. As future work, we aim to write the code of Scintillae for open source platforms, and adding further capabilities to make easier its usage. For instance, it would be very useful to exploit configurations already defined as black boxes for new configurations. This would expand the visual programming features of the game. Moreover, educational characteristics could also be exploited. In fact, Scintillae turns out to be a very good mean for experiencing sequential circuits, but also for an easy approach to low level programming languages like assembly.

Historical Note and Acknowledgement. A first version of Scintillae has been implemented by Gabriele Di Stefano on a PC Olivetti M24 in the mid-eighties. Special thanks go to Gian Marco Tedesco for his great contribution in coding Scintillae with graphic libraries, for his insights and useful discussions.

References

1. `http://www.dmi.unipg.it/navarra/Scintillae/scintillae.zip`
2. `http://gs.ing.univaq.it/Scintillae/scintillae.zip`
3. Chang, S.K.: Visual Languages and Visual Programming (Languages and Information Systems). Springer (1990)
4. Adamatzky, A.: Game of Life Cellular Automata. Springer (2011)
5. Gardner, M.: The fantastic combinations of John Conway's new solitaire game "Life". Scientific American 223, 120–123 (1970)
6. `http://www.bitstorm.org/gameoflife/`
7. Kari, J.: Theory of cellular automata: a survey. Theoretical Compututer Science 334, 3–33 (2005)
8. Wolfram, S.: A New Kind of Science. Wolfram Media, Inc. (2002)
9. Dewdney, A.K.: Computer recreations: The cellular automata programs that create wireworld, rugworld and other diversions. Scientific American 262, 146–149 (1990)
10. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Elsevier Inc. (2007)