Bacharelado em Ciência da Computação

Rodrigo Farias de Figueiredo

GENESIS: PLATAFORMA DE MODELAGEM E SIMULAÇÃO DE AUTÔMATOS CELULARES

Trabalho de Graduação

Universidade Federal de Pernambuco
secgrad@cin.ufpe.br
www.cin.ufpe.br/~secgrad

RECIFE

Junho, 2017

Universidade Federal de Pernambuco
Centro de Informática
Bacharelado em Ciência da Computação

Rodrigo Farias de Figueiredo

GENESIS: PLATAFORMA DE MODELAGEM E SIMULAÇÃO DE AUTÔMATOS CELULARES

Trabalho apresentado ao Programa de Bacharelado em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Tsang Ing Ren*

RECIFE
Junho, 2017

Rodrigo Farias de Figueiredo
Genesis: Plataforma de Modelagem e Simulação de Autômatos Celulares/ Rodrigo Farias de Figueiredo. – RECIFE, Junho, 2017-
59 p. : il. (algumas color.) ; 30 cm.

Orientador Tsang Ing Ren

Trabalho de Graduação – Universidade Federal de Pernambuco, Junho, 2017.

1. Palavra-chave1. 2. Palavra-chave2. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

CDU 02:141:005.7

Aos que acolhem a tempestade sem hesitar,
e por ela passam sem se molhar

Agradecimentos

Esta parte do trabalho é especialmente relevante quando o assunto envolve emergência. Isto porque seria no mínimo contraditório promover o uso de autômatos celulares - em cuja lição essencial é a de que não subestimemos o impacto das coisas simples -, e tratar com levianidade a reflexão sobre todas as coisas externas a minha vontade que me impeliram até este momento.

Não se trata de lembrar daqueles que direta ou indiretamente investiram energia neste documento, no projeto ou em mim. Sequer tenho essa capacidade, e me angustiaria se tentasse fazê-lo. Tampouco chamar a atenção para qualquer que seja minha crença religiosa. Em vez disso, se trata de contemplar por um instante a profundidade de nossa própria ignorância, que nos encoraja a acreditar em uma realidade rasa, onde nossa percepção limitada é suficiente.

Quando criança, meus pais com frequência meditavam comigo e meus irmãos, sobre como passamos desapercebidos da energia e esforço investidos por incontáveis pessoas, para que uma dada refeição estivesse então posta em nossa frente. A conclusão era sempre a mesma: quanto mais refletíamos, maior o número de pessoas envolvidas. Desde associações imediatas como meus pais, até aqueles que participaram da concepção das ferramentas utilizadas pelo agricultor, no terreno que produziu os grãos em nossos pratos. Percebíamos com isto, que, de forma similar, também influenciamos muito mais pessoas do que somos capazes de perceber ao longo da nossa vida, através do espaço e tempo. Esse é um modo de compreender que por meio de nossas ações, benéficas ou não, nos conectamos a todos, pelo simples fato de partilharmos o mesmo ambiente. Isto foge ao nosso controle e encoraja mais responsabilidade no que fazemos.

Sendo assim, reconheço minha atual incapacidade de ser preciso e acurado em meus agradecimentos. Sei que até mesmo o gesto mais distante e sem intenção, de pessoas que nem mesmo conheço, podem determinar importantes acontecimentos em minha vida. É um exercício infinito de evocação das causas que precedem as causas.

A lembrança mais antiga do meu interesse por fenômenos emergentes remete a quase uma década atrás, quando sobre um certo cajueiro (que já não existe) meu irmão conversou comigo sobre vida artificial. Isto despertou a curiosidade necessária para que anos depois, numa aula de Informática e Sociedade, eu tivesse proatividade suficiente para questionar o professor Clylton sobre os autômatos celulares mencionados na sala. Ele então me deu incentivo e material para que eu me aprofundasse no assunto. Posteriormente, na disciplina de Metodologia Científica, a professora Patrícia me estimulou a continuar o desenvolvimento do *Genesis*, após a apresentação de um singelo protótipo. Mais adiante, numa conversa improvável com o professor Tsang e seu irmão, encontrei um orientador que já havia se aprofundado no tema e partilhava dos meus interesses. Assim, recobrei o ânimo necessário para lapidar as ideias e dar seguimento ao projeto.

Por isso, sinto profunda gratidão geral. Afinal, todos que exerceram uma influência - por menor que tenha sido - na minha existência, contribuíram para o que me tornei *aqui e agora*.

A intuição é uma das maiores facetas humanas. Você intui,
e mesmo que não seja do nada, não se sabe de onde...
mas que intui, intui.

—CLYLTON GALAMBA

Resumo

Autômatos celulares compõem uma classe de modelos comumente utilizados na simulação e análise de sistemas complexos e dinâmicos, sendo aplicáveis a diversas áreas e problemas, tais como: processamento de imagens, espalhamento de doenças e simulações físicas. Existem diversos programas e plataformas que buscam oferecer suporte à modelagem e simulação de autômatos celulares. No entanto, enquanto a maior parte deles se restringe a modelos já conhecidos ou permite pouca alteração, outra parte oferece a possibilidade da criação de novos modelos apenas por meio de linguagens de programação tradicionais (como *java* e *XML*) ou específicas da plataforma. Esse trabalho tem como propósito o desenvolvimento do *Genesis*: uma plataforma capaz de oferecer uma interface intuitiva para usuários que desejem projetar ou avaliar seus próprios autômatos celulares. Por meio do uso de uma linguagem de programação visual baseada em componentes conectáveis, o usuário pode definir quaisquer tipos de regras, mantendo-se focado na lógica em detrimento dos detalhes estruturais de um código. Além disso, o *Genesis* suporta a exportação do modelo desenvolvido como um código independente em *C++*, facilitando a inclusão no código fonte de outras aplicações. Por fim, uma aplicação executável independente com suporte a visualização e interação também pode ser exportada, permitindo um compartilhamento livre da instalação do *Genesis*.

Palavras-chave: Autômatos Celulares, Ambiente de Desenvolvimento Integrado, Programação Visual, Sistemas Complexos

Abstract

Cellular automata is a class of models commonly used in simulation and analysis of complex and dynamic systems, with applications on numerous problems and fields, such as image processing, spread of diseases and physics simulation. There are a variety of software and environments aiming the support of cellular automata modelling and simulation. However, while most of them are limited to built-in models, with small or no change allowed, the rest imposes the use of traditional programming languages (e.g. *java* and *XML*) or environment specifc ones to define new models. This work presents the development of *Genesis*: an software capable of providing an intuitive interface for users who wish to design and evaluate their own cellular automata models. By using a visual programming language based on connected components, the user is able to define any kind of rules, while keeping focused on model logic instead of structural details of coding. Besides, the *Genesis* supports exportation of developed cellular automata as an independent *C++* code, allowing other applications to incorporate the model in their source code. Finally, its possible to export a standalone application with its own visualization and interaction tools, allowing the sharing without requiring the instalation of *Genesis*.

Keywords: Cellular Automata, Integrated Development Environment, Visual Programming, Complex Systems

Lista de Figuras

Lista de Acrônimos

Sumário

1

Introdução

1.1 Contexto

Autômato Celular, ou *Cellular Automata (CA)*, é uma classe de modelos discretos no espaço e tempo, tradicionalmente determinísticos, em cujas partes interagem localmente e por definição evoluem em paralelo ILACHINSKI (2001). Em termos práticos, são formados por uma grade de células em estados específicos, que variam ao longo do tempo com base nos estados das células vizinhas. Intuitivamente, CAs são bons exemplos de sistemas matemáticos compostos por muitos componentes idênticos onde, embora individualmente simples, formam um coletivo capaz de expressar comportamentos complexos.

O interesse científico e popularização dos CAs têm início a meio século atrás, quando no fim dos anos 40 John von Neumann procurava meios de projetar um sistema artificial auto replicativo. Junto a Stanislaw Ulam ele encontrou nos autômatos celulares a base teórica para suas investigações, resultando na concepção do *construtor universal de von Neumann*: um CA de 29 estados que replica a própria configuração indefinidamente (ver Figura 1.1).

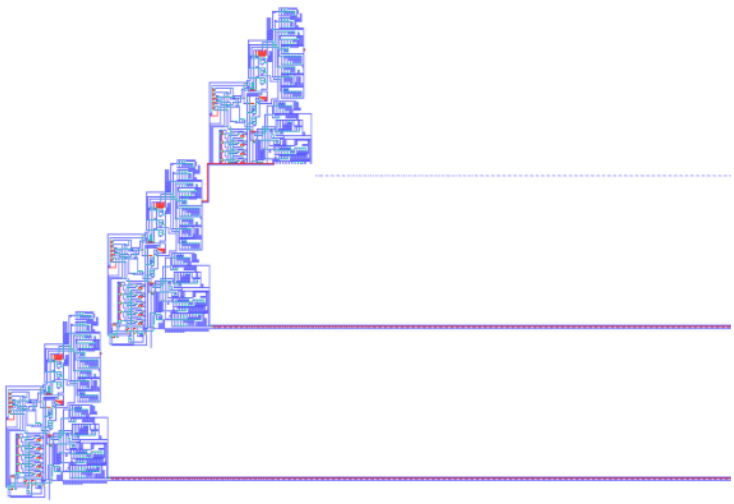


Figure 1.1: Execução do construtor universal de von Neumann. A imagem está em uma escala muito maior que a das células, de modo que seja possível observar as duas réplicas completas (embaixo) e a terceira em construção.

Do ponto de vista de *Computação Natural*, CA se enquadra em um dos três principais ramos do campo: *simulação de fenômenos naturais em computadores*, na medida em que já se mostrou aplicável em diversas situações. Dentre elas, a simulação de um ecosistema com várias espécies de mexilhões WOOTTON (2001) (ver Figura 1.2); do crescimento urbano WHITE; ENGELEN (1993); das complexas interações entre diferentes genes QIU et al. (2014); e do espalhamento de doenças transmitidas por vetores (dengue) MELLO; CASTILHO (2014);

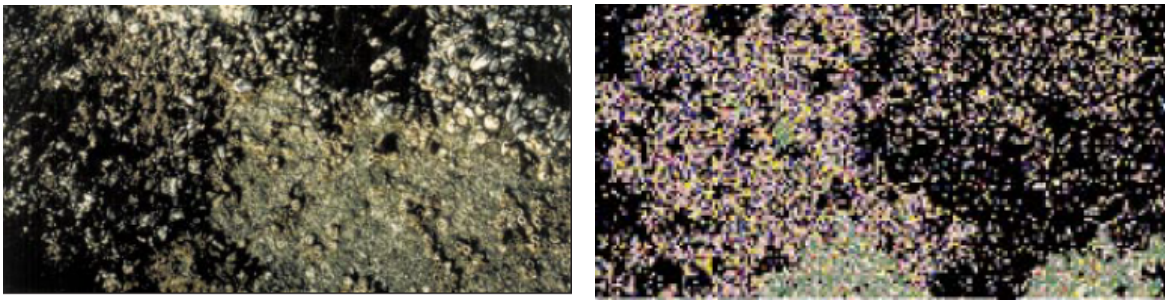


Figure 1.2: Resultado da simulação do ecosistema de mexilhões, apresentado em WOOTTON (2001). Na esquerda, fotografia dos padrões espaciais de um *habitat* de mexilhões em Washington. Na direita, padrões simulados usando o modelo de CA.

No campo de *Processamento de Imagens*, CAs expandem as possibilidades de filtros, uma vez que podem ser vistos como convoluções arbitrariamente complexas em detrimento dos clássicos e restritos *operadores morfológicos*. Isto possibilita ajustes precisos do comportamento desejado, mas deve-se destacar que também impõe mais responsabilidades ao projetista. As aplicações incluem o compartilhamento de imagens secretas com esteganografia [1] ESLAMI; RAZZAGHI; AHMADABADI (2010); detecção de imagens fraudulentas (regiões duplicadas) TRALIC et al. (2014); segmentação de tumores em imagens de ressonância magnética HAMAMCI et al. (2010); remoção de ruídos SELVAPETER; HORDIJK (2009) (ver Figura 1.3); detecção de arestas CHANG; ZHANG; GDONG (2004); e até mesmo a geração de imagens ao estilo *Suibokuga*[2] ZHANG et al. (1999).

Figure 1.3: Resultado da remoção de ruído impulsivo na imagem Lena, apresentado em SELVAPETER; HORDIJK (2009). Na esquerda, a imagem com 20% de ruído. Na direita, resultado após aplicação do CA proposto.

[1] Ramo da criptografia em que uma mensagem é ocultada utilizando outra mensagem

[2] Suibokuga é uma uma técnica de pintura oriental do século II essencialmente monocromática

Por fim, existem outras inúmeras aplicações distribuídas nos mais variados contextos, como em Teoria da Informação: criptografia WOLFRAM (1985) e compressão de dados LAFE (1997); Informática Teórica: modelos para computação quântica AMLANI et al. (1999); Física: simulação do fenômeno de recristalização RAABE; BECKER (2000) e simulação de fluidos WOLF-GLADROW (2004) (ver Figura 1.4); Teoria dos Jogos: geração procedural de cenários JOHNSON; YANNAKAKIS; TOGELIUS (2010) (ver Figura 1.4); Música: síntese artificial de músicas MIRANDA (2001).

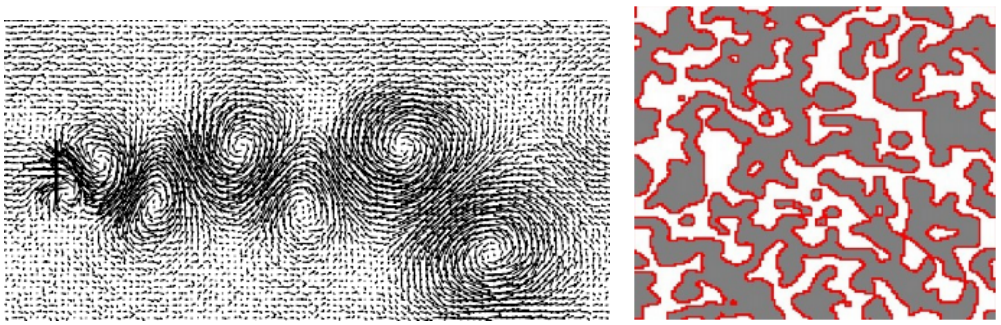


Figure 1.4: Exemplos de outras aplicações. Na esquerda, imagem de um experimento da simulação de fluidos usando CAs proposta em WOLF-GLADROW (2004). Na direita, um cenário de cavernas gerado pelo CA proposto em JOHNSON; YANNAKAKIS; TOGELIUS (2010).

1.2 Motivação

Visto que a teoria de autômatos celulares enriquece as mais diversas áreas da ciência, é de se esperar que hajam diversas ferramentas de auxílio à concepção e experimentação de novos CAs. Porém, o que se observa de fato é uma carência de opções que ofereçam liberdade de criação sem sacrificar a acessibilidade aos diferentes tipos de usuários -em especial aos leigos em computação. Isto motivou o desenvolvimento do presente trabalho, em cuja abordagem se baseia numa plataforma de modelagem visual que contempla as restrições de leigos e cientistas com limitado conhecimento e/ou experiência em linguagens de programação.

1.3 Fundamentação Teórica

Em cada forma distinta de aplicação, é natural que existam necessidades diferentes na modelagem, execução e até mesmo na definição dos autômatos, portanto, é possível encontrar diversas extensões da definição original. Isto pode gerar certa confusão, especialmente aos que desejem utilizar um modelo de CA pela primeira vez. ILACHINSKI (2001) apresenta uma definição abrangente, que captura a essência do modelo, definindo que todo CA deve possuir as seguintes características:

- *Uma grade discreta de células* – o sistema é definido por uma grade unidimensional, bidimensional ou n-dimensional. Em que cada célula pode ter diferentes formas, como triângulos, quadrados ou hexágonos.

- *Um conjunto finito de estados discretos* – em cada intervalo discreto do tempo, cada célula deve estar em apenas um dos estados permitidos.
- *Interações locais* – cada célula é capaz de adquirir informação apenas de células em sua vizinhança. A vizinhança pode assumir qualquer forma, contanto que respeite-se a noção de localidade; *i.e.*, não é coerente que células sejam capazes de perceber grandes porções da grade.
- *Dinâmicas discretas* – A cada intervalo discreto do tempo, todas as células atualizam seus estados de forma paralela. Isto é feito de acordo com uma regra de transição, que define o próximo estado da célula com base no estado anterior da própria célula e dos seus vizinhos.

Definição Formal

Uma definição mais clássica, do livro *Fundamentals of Natural Computing* DE CASTRO (2006), declara um CA C como a 5-tupla

$$C =< S, \mathbf{s}_0, G, d, f > \tag{1.1}$$

onde S é um conjunto finito de estados, $\mathbf{s}_0 \in S$ são os estados iniciais das células, G é a vizinhança observável, $d \in \mathbb{Z}^+$ é a dimensão de C, e f é a *função de transição* ou *regra de transição*.

Foi observado na literatura uma necessidade, e prática comum, em relaxar as definições formais, visando modelar o CA com menos restrições. Isto inclui definições que apenas expandem a definição clássica, como utilizar um conjunto em vez de uma única vizinhança MELLO; CASTILHO (2014); mas também algumas que contrariam alguns princípios fundamentais da teoria, como permitir que uma célula imponha alterações nas informações de outra célula AVOLIO et al. (2003).

Como tal discussão foge ao nosso escopo, daqui em diante adotaremos uma definição próxima da enunciada no início do capítulo. Para nós, as seis características essenciais são:

1. Células são agentes com livre poder computacional
2. Células possuem atributos internos, em cujos valores denominamos *configuração*
3. Células possuem percepção restrita à lista *constante* de vizinhanças pré-definidas
4. Células atuam apenas sobre sua própria configuração, nunca diretamente no ambiente
5. Espaço e Tempo variam de forma discreta
6. Todas as células atualizam suas configurações simultaneamente

De modo geral, este é o principal *denominador comum* entre os variados formatos de CAs encontrados na literatura. Ao adotá-lo, estaremos atendendo o máximo de usuários.

1.4 Problema

Dada a diversidade de uso, bem como de comportamento que um CA possui, as abordagens e ferramentas disponíveis para se projetar ou experimentar CAs costumam sacrificar a acessibilidade em função da capacidade de modelagem, e vice-versa. Como consequência, o que se encontra são opções que oferecem uma lista pré-definida de modelos parametrizáveis ou exigem conhecimento específico do usuário.

1.5 Objetivo

Este trabalho tem por objetivo *definir* e *implementar* um ambiente de desenvolvimento que suporte a modelagem e simulação de novos CAs, acessível a usuários sem conhecimentos técnicos específicos. Adicionalmente, será desenvolvida uma linguagem de programação visual por meio da qual será possível manipular irrestritamente o comportamento das células, bem como um módulo de simulação independente com suporte às interações com um modelo pronto.

1.6 Estrutura Do Trabalho

Os capítulos que se seguem estão distribuídos da seguinte maneira: no Capítulo 2 será feita uma análise dos trabalhos relacionados da literatura, incluindo as opções restritas e irrestritas de modelagem; o Capítulo 3 apresenta a solução proposta, suas capacidades, a organização geral da solução, uma explicação sobre a linguagem visual desenvolvida, e ainda alguns fluxos de trabalho possíveis; o Capítulo 4 descreve os detalhes técnicos da solução, a arquitetura dos módulos, abordagens e tecnologias utilizadas; no Capítulo 5 são apresentados os experimentos com alguns casos de uso e uma análise inicial de desempenho; por fim, são expostas as considerações finais e trabalhos futuros na Seção 6.

2

Trabalhos Relacionados

Em seguida serão apresentadas abordagens da literatura que um usuário pode adotar afim de desenvolver ou experimentar modelos de CAs. Optamos por uma classificação em dois grupos: *Modelagens Restritas*, que contempla as opções em que o usuário está de alguma forma limitado na construção das regras; e *Modelagens irrestritas*, que abrange as opções em que o usuário pode definir regras livremente.

2.1 Modelagens Restritas

As opções que seguem alguma das condições abaixo são compreendidas como restritas:

- Se limitam a simular modelos pré-definidos
- Exigem que o CA construído respeite determinados formatos de regras

Encontram-se neste grupo as abordagens que um usuário pode adotar a fim de experimentar ou construir regras sem abrir mão da acessibilidade, i.e. sem conhecimento prévio específico. Foi observado que isto inclui principalmente programas de caráter educativo, que focam em fornecer uma intuição sobre o modelo de CA, bem como os mais especializados na visualização e interação com a simulação em si. Ou seja: de modo geral oferecem maior acessibilidade mas restringem as opções do usuário. Vale salientar que optamos por incluir neste grupo alguns programas que, embora permitam a simulação de regras personalizadas, oferecem um modo de fazê-lo ou linguagem restritivos.

Um dos mais antigos (1995) mas que permanece sendo atualizado e utilizado até hoje em experimentos científicos é o *DDLab* WUENSCHE (1996). Embora ofereça diversos tipos de modelos, visualizações e análises, este programa falha em dois pontos cruciais sob a ótica do nosso contexto: usabilidade e poder de modelagem. Isto porque seu meio de interação principal é ao estilo *console*, com diversos atalhos para todos os tipos de interação; e para definir um CA é preciso fazer uso de *tabelas de regras*[1]. Tal método de modelagem restringe não apenas que tipo de informação uma célula pode conter, como a interpretação clara do tipo de comportamento que uma determinada tabela significa (ver Figura 2.1).

[1] Tabelas de regras são usadas para mapear configurações da vizinhança em estados de uma célula

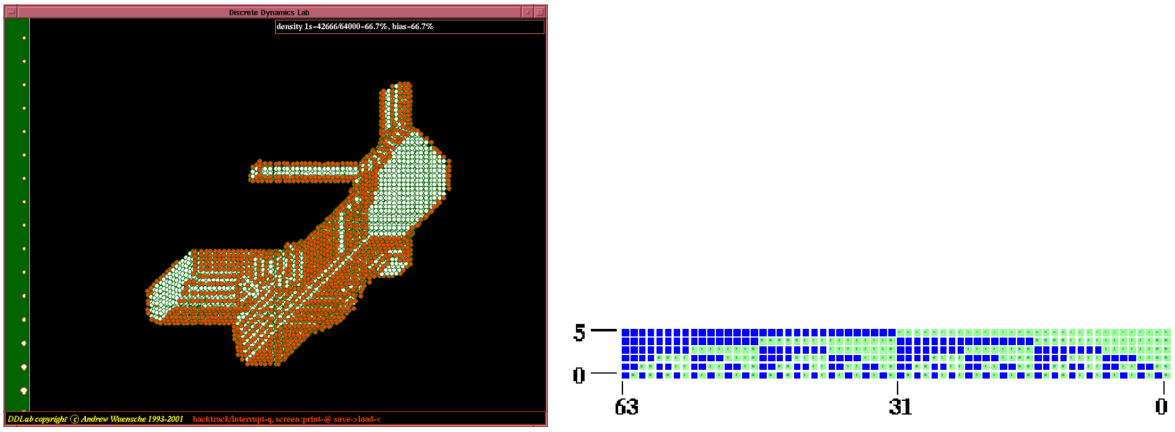


Figure 2.1: Programa *DDLab*. Na esquerda, a interface de execução. Na direita, um exemplo de tabela de regras onde a célula só pode assumir duas configurações possíveis (verde e azul) com vizinhança de von Neumann. Cada coluna representa uma configuração de vizinhança e como a célula deve reagir.

NetLogo (TISUE; WILENSKY (2004)) é um ambiente de modelagem de multi-agentes programáveis, onde é possível construir e simular sistemas emergentes. Dentre outros propósitos, seu público alvo inclui educadores e especialistas de outras áreas para que possam investigar este tipo de fenômeno. O *NetLogo* é considerado *restrito* na medida em que limita os usuários leigos à experimentação de modelos existentes, já que exige o conhecimento prévio de sua linguagem própria para a criação personalizada de novos modelos (ver Figura 2.2).

É relevante destacar que a interação com os parâmetros do modelo por meio de componentes gráficos é uma característica desejável, permitindo maior *insight* sobre como as propriedades internas influenciam o comportamento global, e facilitando assim o processo experimental de concepção. Destacamos também a possibilidade de definir mais de uma inicialização, que pode ser crucial para a construção de modelos sensíveis a configurações iniciais específicas (e.g. o supracitado *construtor universal* da Figura 1.1).

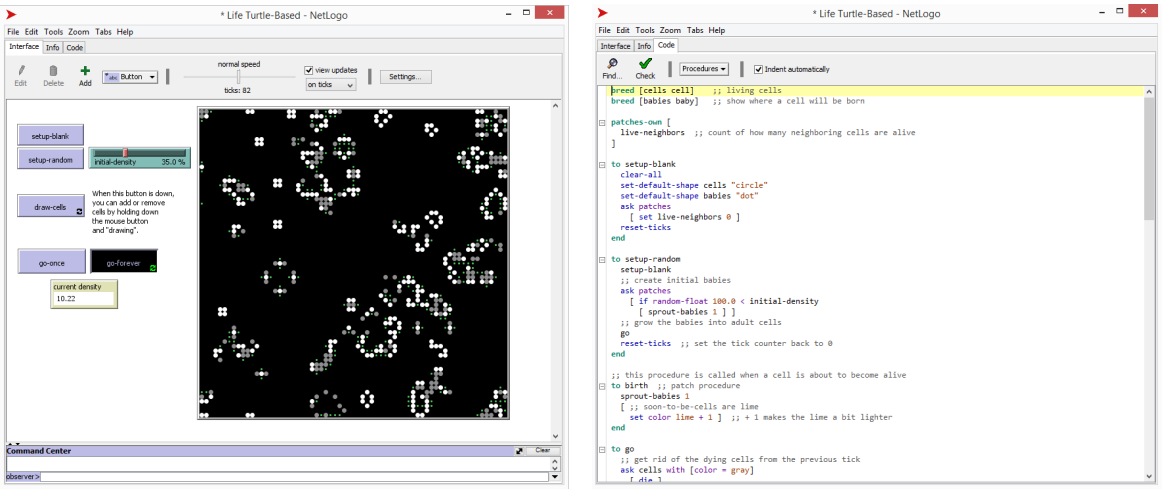


Figure 2.2: Programa *NetLogo*. Na esquerda, a tela de execução com os parâmetros ajustáveis via itens interativos. Na direita, o código do modelo em execução (*Game of Life*).

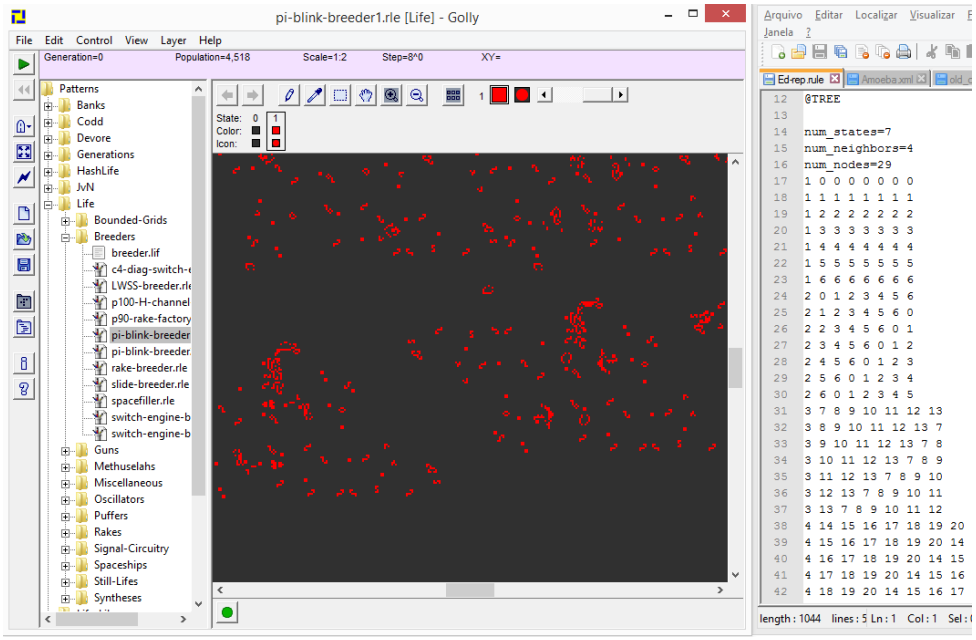


Figure 2.3: Programa *golly*. Na esquerda, temos a interface principal. Na direita é exibido um código usado para descrever as regras interpretadas pelo *golly*.

Um programa menos popular, mas ainda muito difundido é o *golly*. Com centenas de modelos pré-definidos em sua biblioteca, é uma opção recomendada para os usuários que buscam experimentar os modelos existentes. Porém para o nosso contexto, o *golly* falha ao fornecer apenas variações de *tabelas de regras* para a definição de novos modelos (ver Figura 2.3).

Exitem outros modelos menos populares, como o *CASim* que restringe as regras geradas ao formato <*estado anterior*> *([estados dos vizinhos])* -> <*novo estado*>, e restringe a informação contida em uma célula a um número inteiro (ver Figura 2.4).

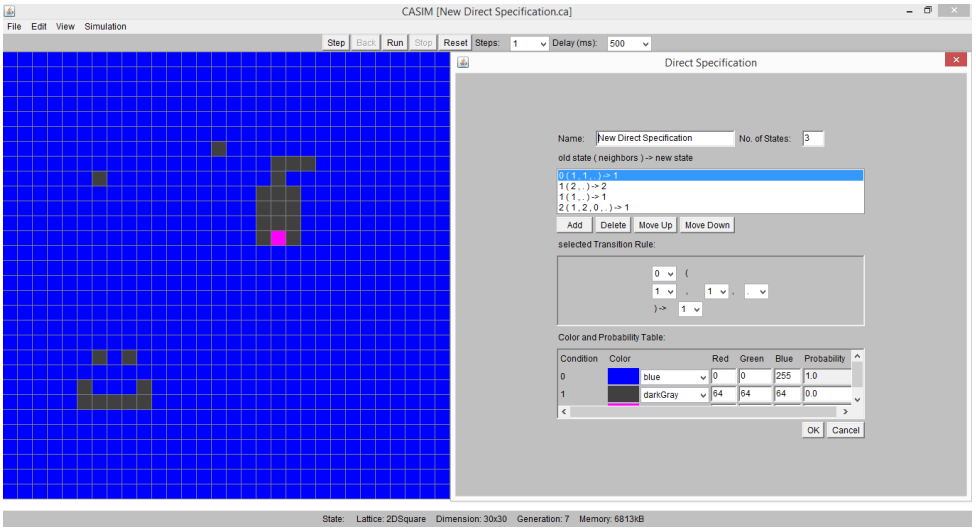


Figure 2.4: Programa *CASim*. Interface principal com janela de definição de regras na direita.

Dentre os muitos outros (*CelLab*, *Mcell*, *JCASim* (FREIWALD; WEIMAR (2001)), *CaFun*, *Life32*) que oferecem um conjunto pré-definido de modelos (alguns adicionalmente com

criação de regras customizadas por formatos não intuitivos), destacamos o *CaFun*. Embora também não se ajuste aos propósitos deste trabalho, ele possui uma característica desejável: permitir visualizações independentes da configuração das células (ver Figura 2.5). Isto concede ao usuário a capacidade de observar o modelo em execução sob diversas perspectivas, bem como separar as informações pertinentes à lógica das que se deseja apresentar.

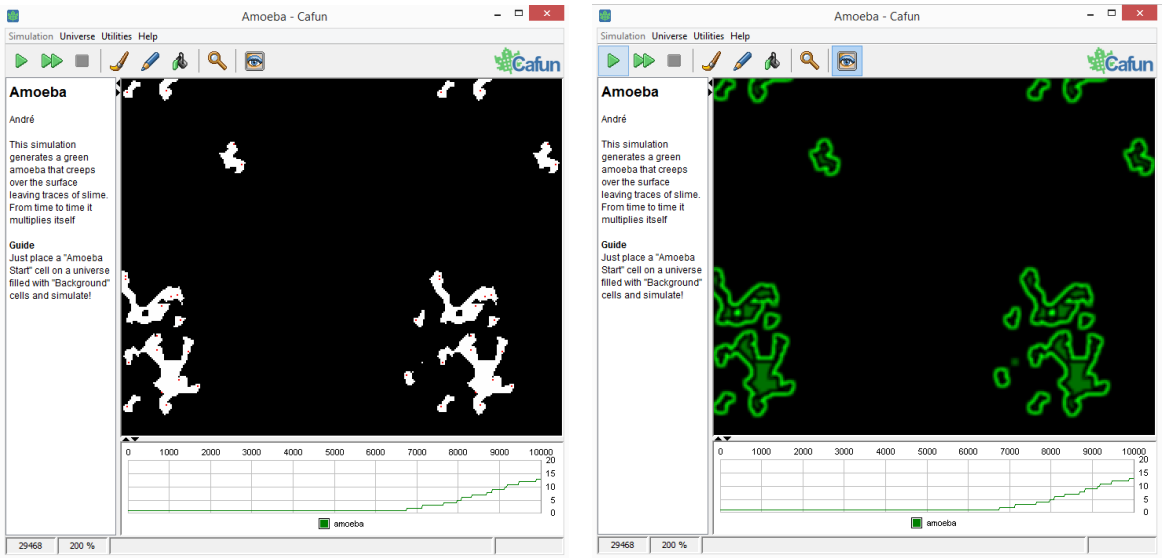


Figure 2.5: Programa *Cafun*. Do lado esquerdo a visualização padrão é exibida, onde é possível observar o modo pelo qual ele simula a *ameba*, com uma célula central (ponto vermelho) para cada parte, rodeada da região de *"limo"*. Na direita é possível observar uma visualização mais agradável e objetiva na extensão da ameba, que é o propósito da simulação.

2.2 Modelagens Irrestritas

As opções com liberdade para a criação de regras mas que seguem alguma das condições abaixo são compreendidas como irrestritas:

- Exigem o conhecimento de linguagens específicas e/ou pouco intuitivas
- Exigem que o usuário construa seu próprio *framework* para implementar os CAs definidos e/ou testá-los

Encontram-se neste grupo as abordagens que um usuário pode adotar a fim de construir CAs ajustados aos mais específicos propósitos. Será visto que para isso a acessibilidade acaba sendo sacrificada.

Programas como *Jcell*, *CAGE*, e o já mencionados *JCASim* e *NetLogo*, permitem a definição de regras adaptadas aos interesses do usuário. Porém, é exigido que o usuário não apenas tenha conhecimento e experiência na linguagem permitida (Java, C++, NetLogo), como também conheça o que é permitido em cada programa (ver Figura 2.6).

É crucial destacar que, como o código da regra é integrado diretamente ao programa, torna-se obrigação do usuário respeitar as convenções do sistema. Ou seja: definições arbitrárias

como estruturação e nome das classes, interfaces de funções, tipos de vizinhanças e dados permitidos para cada célula. Embora irrelevante ao modelo do usuário, tudo isto deve ser considerado durante a concepção do CA para o correto acoplamento com o programa.

```
class ConwaysLife extends GenericRuleset
{
    public int[][] doGeneration(Board current)
    {
        board = current.board;
        int[][] newBoard = new int[board.length][board[0].length];

        for(int i = 0; i < width; i++)
        {
            for(int j = 0; j < height; j++)
            {
                int count = RulesetUtil.getEightCount(i, j, width, height, board, wrapping);
                if (count == 2)
                {
                    newBoard[i][j] = board[i][j];
                }
                else if (count == 3)
                {
                    newBoard[i][j] = 1;
                }
```

```
<abstract-cell-type id="floating">
  <abstraction id="$North_West"/>
  <abstraction id="$North"/>
  <abstraction id="$North_East"/>
  <abstraction id="$East"/>
  <abstraction id="$South_East"/>
  <abstraction id="$South"/>
  <abstraction id="$South_West"/>
  <abstraction id="$West"/>
  <mutation cell-type="$North_West">
      <condition cell-type="Amoeba_South_East" min="1" scope="north-west"/>
  </mutation>
  <mutation cell-type="$North">
      <condition cell-type="Amoeba_South" min="1" scope="north"/>
  </mutation>
  <mutation cell-type="$North_East">
```

Figure 2.6: Trechos de códigos usados na definição de CAs. Acima, classe *Java* do CA *Game of Life* lida pelo programa Jcell, com algumas convenções arbitrárias de nomenclatura sublinhadas em vermelho. Abaixo, trecho do código *XML* que define o CA *Amoeba* (exibido na Figura 2.5), com algumas convenções sublinhadas em preto.

Um usuário que já soubesse ou estivesse disposto a aprender uma linguagem de programação para modelar CAs, e não desejasse se submeter às convenções dos programas de terceiros, deveria para isso abrir mão das facilidades que estes programas trazem, e implementar tudo. Ou seja: a infra-estrutura comum a todos os CAs, incluindo os mecanismos de atualização das células; referências para as vizinhanças e seus estados anteriores; formas de visualizar a execução; interações com a simulação em curso; leitura de configurações iniciais; e funcionalidades periféricas como análises estatísticas da configuração, bem como salvar imagens instantâneas.

Fazem parte desta abordagem trabalhos como: a simulação do espalhamento da Dengue MELLO; CASTILHO (2014), estudo do crescimento de folhas YATAPANAGE (2003), simulação do escoamento de lava AVOLIO et al. (2003) e a análise de padrões criminais LIANG (2001).

Um programa em particular que permite uma modelagem irrestrita deve ser destacado: *StarLogo* KLOPFER et al. (2009). Este, assim como o supracitado *NetLogo*, é um ambiente de programação criado para explorar as dinâmicas de sistemas descentralizados (ver Figura 2.7). O que chama a atenção é a acessibilidade do projeto, que utiliza uma linguagem visual para a definição dos modelos. Livrando desta forma a necessidade de conhecimento prévio do usuário, como ilustrado na Figura 2.8.

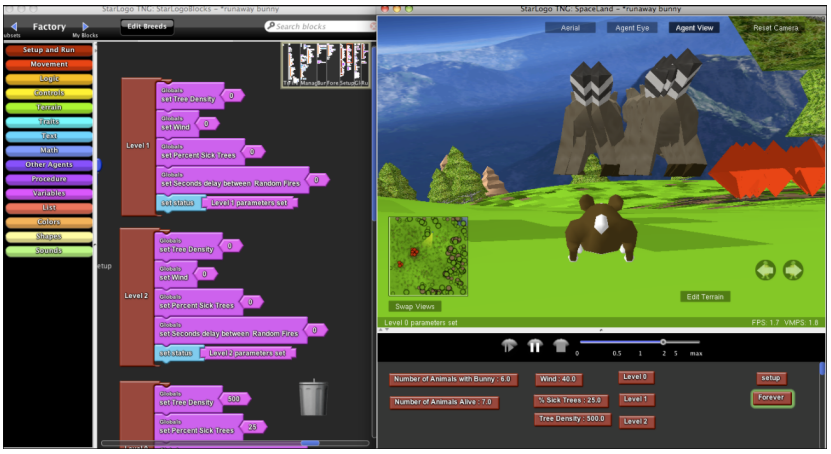


Figure 2.7: Inteface do programa *StarLogo*. Definição do comportamento dos objetos no lado esquerdo e simulação na janela superior direita.

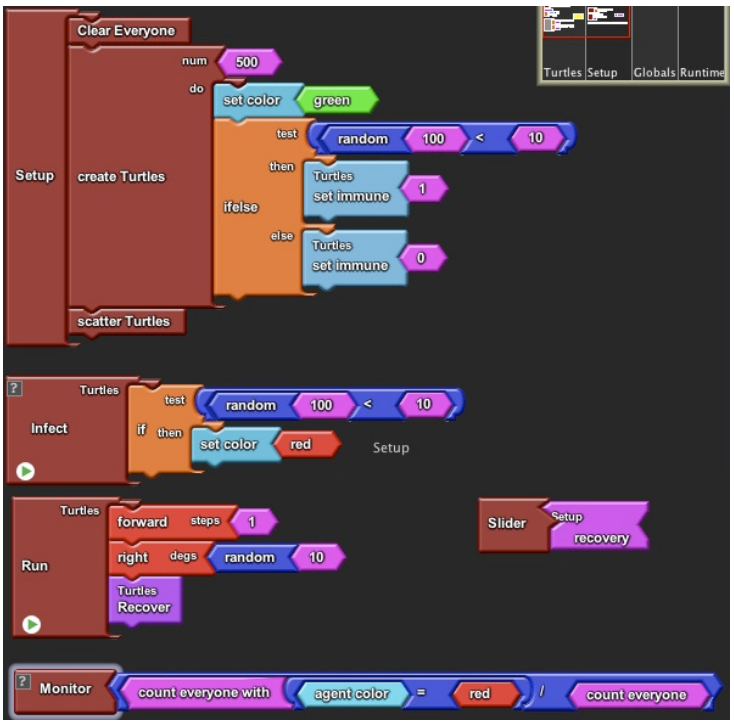


Figure 2.8: Liguagem visual do programa *StarLogo*. Assim como um quebra-cabeça, a linguagem é formada de inúmeros blocos de encaixar. As formas e cores são importantes para diferenciar os tipos e funções dos blocos.

Infelizmente, o universo de modelos permitidos pelo *StarLogo* é muito maior que o dos CAs, criando uma dificuldade natural para usá-lo em nosso contexto. Não se trata de ignorar

funcionalidades do programa e usar apenas o necessário para construir autômatos celulares, mas sim configurá-las de modo a restringir as possibilidades. Por exemplo: num modelo de CA o ambiente é composto pelos próprios agentes (*células*), não havendo distinção; já no *StarLogo*, é necessário definir agentes (*turtles*) e ambiente (*patches*) como entidades separadas.

Dessa forma, embora fazendo uso de uma linguagem acessível e oferecendo a definição de regras irrestritas, o *StarLogo* ainda falha em fornecer uma modelagem intuitiva de novos autômatos celulares.

2.3 Considerações

Existem diversas opções de programas para desenvolver e explorar CAs, no entanto, nenhuma resolve o problema enfrentado em sua completude. Foi observado que de forma geral, quanto maior o suporte ao projetista, menos liberdade na modelagem lhe é oferecido.

É natural que comportamentos não-triviais demandem modelos mais complexos, no entanto, diferentes formas de se apresentar o mesmo modelo podem conduzir a variados níveis de entendimento. Por exemplo, para alguém que desconhece o CA *Game of Life* certamente será mais rápido compreender seu comportamento analisando um fluxograma em detrimento do código em *NetLogo* (ver Figura 2.9).

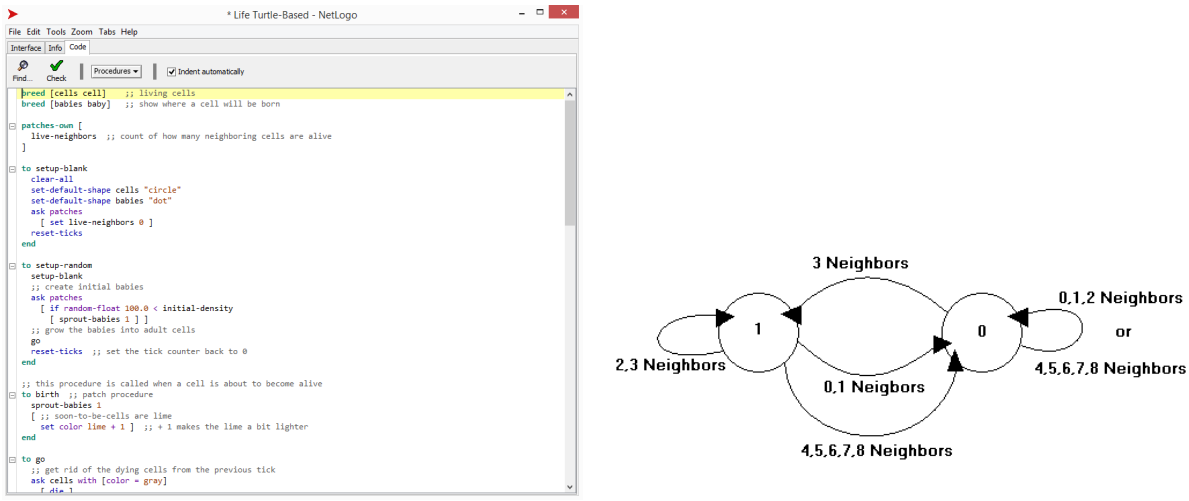


Figure 2.9: Regra *Game of Life* definida em código no programa *NetLogo* e como um fluxograma. Mesmo para conhecedores da linguagem *NetLogo*, a leitura do fluxograma certamente é mais fácil.

Além disso, deve-se destacar que existe um limite de usabilidade no uso de códigos para definir regras complexas. Isto vai além das convenções ou qualidades particulares dos programas apresentados, e está intimamente ligado aos problemas estudados em *Engenharia de Software*. Por exemplo o tamanho do código: o influente engenheiro de software Robert C. Martin (*a.k.a. Uncle Bob*) diz em seu livro: *"The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. [...]"* MARTIN (2009). O código em *NetLogo* do CA da figura acima (considerado dos mais simples) por exemplo, possui 114 linhas.

3

Genesis

Em seguida será apresentado o *Genesis*, a solução proposta ao problema em questão. Este capítulo possui um enfoque nas características e capacidades do programa, e como estas podem ser utilizadas pelos usuários. Uma descrição detalhada da arquitetura, tecnologias e abordagens técnicas será feita no Capítulo 4.

3.1 Apresentação

O *Genesis* é um ambiente de desenvolvimento integrado *(Integrated Development Environment (IDE))*, que oferece suporte a usuários que desejem modelar ou simular CAs irrestritamente, dispensando a manipulação direta de códigos. Em outras palavras, é uma IDE de CAs acessível para leigos em computação (ver Figura 3.1).

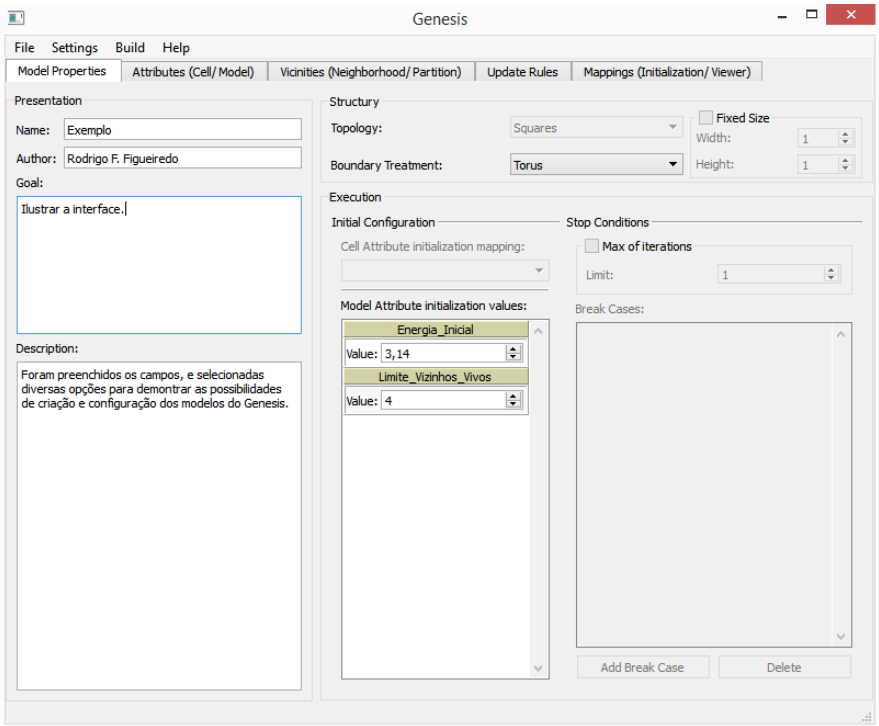


Figure 3.1: Tela inicial do *Genesis*. Todas as configurações básicas são definidas utilizando esta janela de múltiplas abas.

Isto é alcançado através do uso de uma interface intuitiva para a configuração das principais propriedades do CA, e o desenvolvimento de uma linguagem de programação visual *(Visual Programming Language (VPL))* para a definição dos comportamentos específicos. A VPL é compilada internamente para um código C++ de forma completamente transparente ao usuário, permitindo que durante a modelagem o foco esteja na lógica em vez de nos detalhes estruturais do código (ver Figura 3.2).

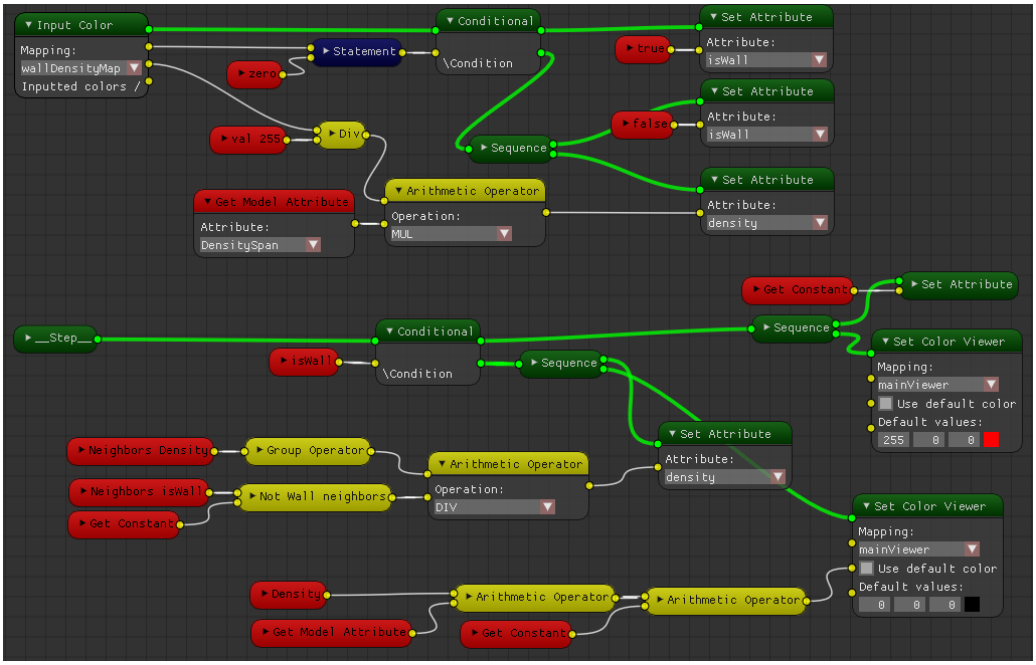


Figure 3.2: Exemplo de regra definida na linguagem de programação visual desenvolvida. As cores desempenham papel crucial na semântica e organização da linguagem.

O *Genesis* permite a exportação de uma biblioteca em C++ (*DLL*) composta pelo modelo projetado, ou do próprio código C++, de modo a facilitar a integração no *pipeline* de projetos de terceiros. Após a inclusão do(s) arquivos gerados e adição de uma única linha ao código, já é possível instanciar e utilizar o modelo livremente (Ver Figura 3.3). Vale salientar que no futuro o *Genesis* pode ser expandido para exportar códigos e módulos em *python* ou *java*, por exemplo.

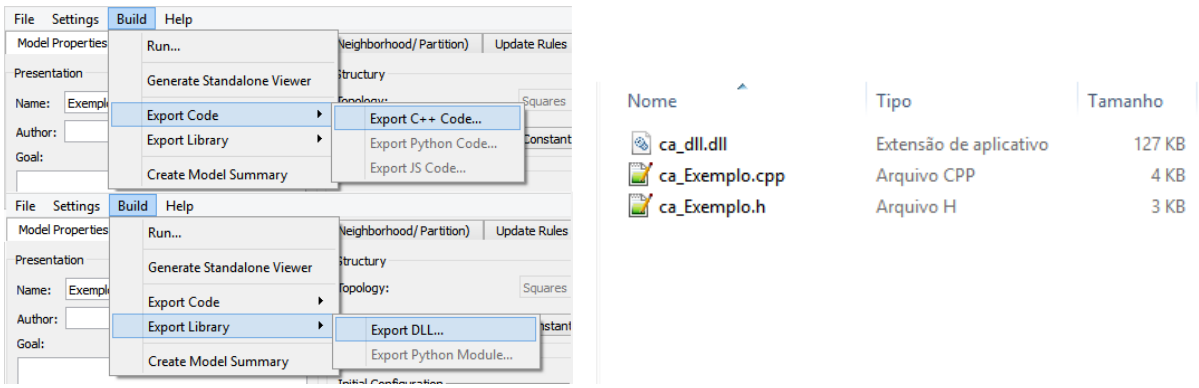


Figure 3.3: Opções de exportação para integração a outros códigos. Em um lado são indicados os locais das opções e no outro encontram-se os produtos das duas possíveis exportações.

Afim de possibilitar a simulação e interação com os modelos gerados, bem como facilitar o posterior compartilhamento de um modelo final, foi criado o módulo *Simulator*. Seu papel é oferecer a interação direta com um modelo em execução e.g. alterar tamanho do CA; parâmetros globais; carregar e salvar imagens; fornecer interação por cliques (ver Figura 3.4).

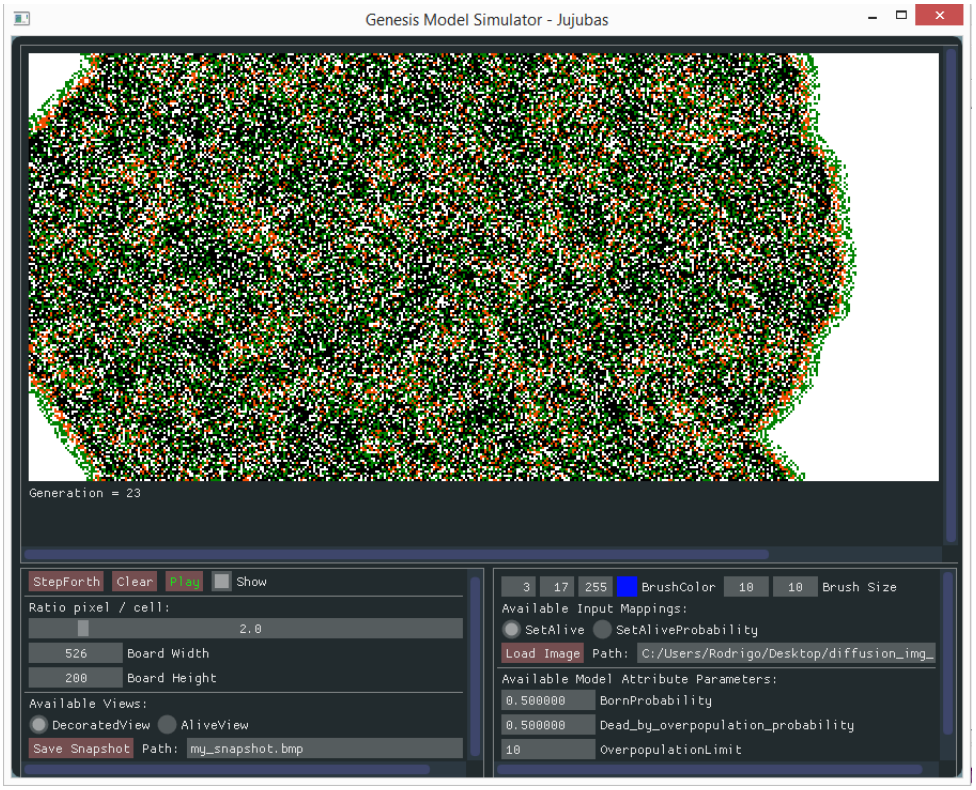


Figure 3.4: Módulo *Simulator* executando um modelo de exemplo. Na parte superior é possível acompanhar a evolução do CA. Na parte inferior, constam as opções relativas à simulação e de interação direta com as propriedades do modelo.

O *Simulator* não apenas serve ao projetista, auxiliando-o na modelagem, como também pode ser utilizado para apresentar e/ou compartilhar o resultado final. Para isso o usuário precisa apenas selecionar a opção de exportar uma aplicação independente. Como resultado, um arquivo executável (*.exe*) é gerado com a mesma interface da Figura 3.4, possibilitando a experimentação *alto-nível* do modelo (ver Figura 3.5).

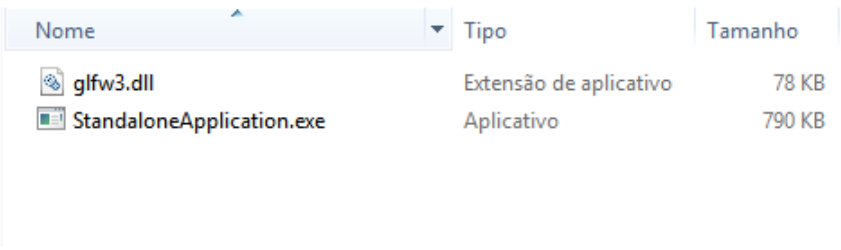


Figure 3.5: Arquivos gerados na exportação da aplicação independente. O executável é independente do *Genesis* ou outros programas. É exigido apenas que a biblioteca gráfica *glfw3.dll* copiada na exportação (usada para desenhar a interface do *Simulator*) esteja na mesma pasta.

3.2 Capacidades

O *Genesis* foi desenvolvido para suportar os diversos modelos de CAs encontrados, com diferentes quantidades e tipos de atributos e vizinhanças; qualquer tipo de comportamento interno das células; definição de parâmetros globais; criação de visualizações arbitrárias; e suporte a diferentes inicializações. Para tanto, fez-se necessário dividir o modelo em:

1. *Propriedades do Modelo* - Configurações globais.
2. *Atributos* - Lista de atributos do modelo e das células.
3. *Vizinhanças* - Lista de vizinhanças utilizadas no modelo.
4. *Regras de Atualização* - Definição dos comportamentos.
5. *Mapeamentos de Cor* - Modos de transformar cores em configurações e vice-versa.

Para cada um dos cinco pontos acima foi criada uma aba na janela principal do *Genesis* (ver Figura 3.6), e os detalhes de organização das opções serão dados na seção subsequente.

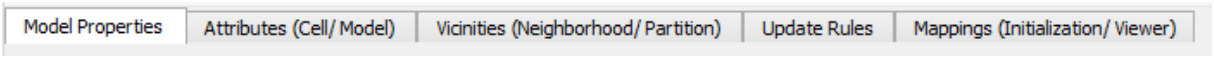


Figure 3.6: Abas da janela principal, usadas para separar logicamente os cinco grupos de opções de configuração do modelo.

Propriedades do Modelo

Configurações como nome, autor, tratamento de bordas, e valores padrão dos atributos de modelo são consideradas propriedades do modelo. Isto porque são opções que, sem a interferência do usuário, devem permanecer constantes ao longo da execução do modelo.

Atributos

São divididos em dois tipos: *atributos de células* e *atributos de modelo*. O primeiro representa informações pertinentes à célula, então chamamos de *configuração* ou *estado* da célula, o conjunto de valores dos seus atributos em um dado instante. Já o segundo representa os parâmetros do modelo. Ou seja: são valores comuns a todas as células, que podem influenciar seus comportamentos.

Por exemplo, no CA que simula o espalhamento da Dengue proposto em MELLO; CASTILHO (2014), foi usado o atributo N_h para indicar o número de humanos na região do mapa representada por uma célula, e o atributo global β_{vh} para parametrizar a probabilidade de transmissão do vetor para o humano (chance da picada ser infeciosa).

Vizinhanças

Para cada CA, ou ainda para diferentes partes do comportamento de um mesmo CA, pode ser útil utilizar diferentes formas de vizinhanças. Isso também é exposto na modelagem defendida por AVOLIO et al. (2003): *"Different elementary processes may involve different neighbourhoods"*. Dessa forma, o *Genesis* permite que uma lista de vizinhanças sem restrição de formato sejam definidas para posterior uso na definição do comportamento.

Regras de Atualização

Todos os comportamentos são definidos nas *regras de atualização*, por meio da linguagem visual. Incluindo a forma como a configuração das células se altera em cada geração, onde e como as diferentes formas de visualização são atualizadas, e ainda o efeito em uma célula ao carregar uma determinada cor. Desta forma, torna-se simples alterar a cor de uma representação da célula em meio à atualização de seus atributos, dando maior contexto às escolhas das cores exibidas (Ver Figura 3.7).

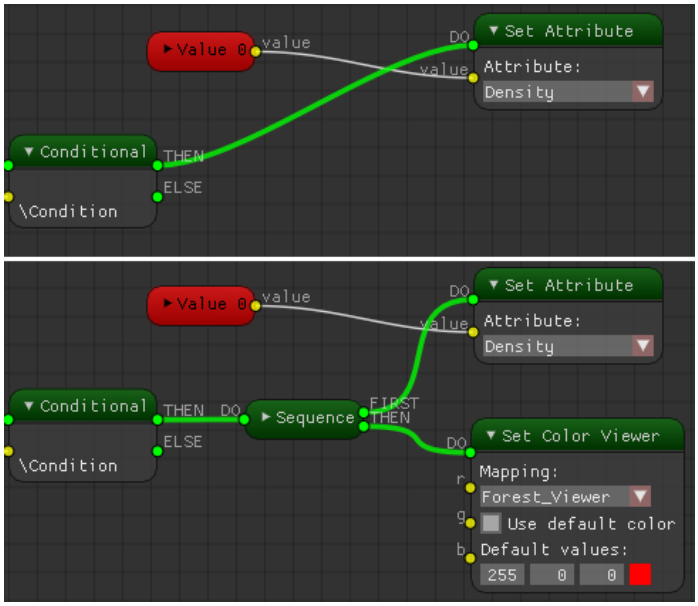


Figure 3.7: Parte de um exemplo hipotético de CA que simule incêndios em florestas. Acima, parte da regra que testa se a célula foi atingida pelo fogo e altera o atributo *Density* (representando a densidade da floresta) para zero. Neste ponto da regra pode ser relevante alterar a cor da célula. Abaixo, destaca-se que para isso, basta uma pequena alteração.

Mapeamentos de Cor

Uma vez que as células não estão restritas a possuírem apenas um atributo, e ainda podem englobar atributos de diferentes tipos (boleano, inteiro, decimal), surgem dois problemas: *como acompanhar a evolução do CA?* e *como interagir ou inicializar os valores dos atributos?* Foi observado que alguns trabalhos relacionados, mesmo que de forma limitada, permitiam diferentes inicializações e formas de visualização padrões. Isto foi aprimorado em nossa proposta.

Para tanto, o *Genesis* oferece os *Mapeamentos de Cor*, que estão divididos em dois tipos: *mapeamentos de saída* (ou *visualizações*), que definem como a configuração de uma célula é exibida em cores; e *mapeamentos de entrada* (ou *inicializações*), que alteram a configuração da célula em resposta a uma determinada cor.

Naturalmente, é possível definir individualmente os atributos de cada célula em um modelo exportado. Porém mapeamentos de entrada podem ser facilitadores cruciais para autômatos mais elaborados. Por exemplo, o *construtor universal* de von Neumann tem como propósito demonstrar a auto-replicação, e para isso depende de uma configuração muito particular das células do tabuleiro, que poderia ser definida com uma imagem. Por sua vez, o CA usado para simular o escoamento de lava AVOLIO et al. (2003), poderia se beneficiar definindo um mapeamento que utiliza o canal vermelho para definir a S_a (altitude do terreno) e o verde para a S_{th} (densidade da lava). Desse modo, seria possível armazenar casos de uso do modelo num formato visivelmente compreensível.

Da mesma forma, é possível ler individualmente os valores dos atributos das células em um modelo exportado. Mas quando se trata de experimentação ou apresentação, matrizes de dados não oferecem grandes *insights*. Por outro lado, visualizar o mesmo CA sob diferentes perspectivas permite ao projetista analisar com mais propriedade características-chave isoladas numa visualização específica.

Durante o desenvolvimento de um CA para detecção de arestas, por exemplo, um projetista poderia sentir a necessidade de analisar a distribuição de células que não foram classificadas como aresta, mas possuem vizinhos que o foram. Ista visualização seria como um modo *debug*, servindo para coletar informações e auxiliar o processo experimental. Outro cenário, seria utilizar uma visualização focada na apresentação do resultado: mais amigável e objetiva no que deve ser exposto, ocultando detalhes do processo (como mostrado na Figura 2.5).

3.3 Organização

Como já dito, a criação e manipulação de modelos no *Genesis* consiste em utilizar as opções oferecidas nas cinco abas da Figura 3.6, bem como a janela do *Simulator*. Em seguida será apresentado como as funcionalidades supracitadas estão organizadas. É importante notar que existem opções desabilitadas na interface, que foram incluídas para suporte em futuras versões do *Genesis*.

Propriedades do Modelo

Nesta aba encontram-se três grupos de propriedades: de apresentação (*Presentation*), estruturais (*Structury*) e de execução (*Execution*), como indicado na Figura 3.8. Na versão atual a topologia é fixada em quadrados (como na maioria dos trabalhos encontrados), porém é possível escolher entre fronteiras constantes ou circulares (respectivamente *constant* e *torus*). Nas propriedades de execução, o projetista pode definir o valor de inicial dos atributos de modelo.

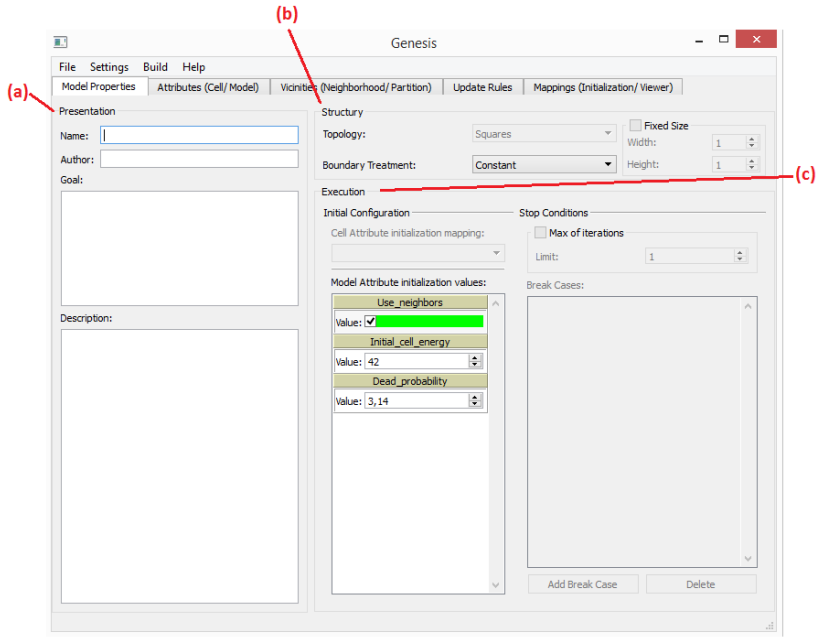


Figure 3.8: Aba de propriedades do modelo. (a) indica as propriedades de apresentação, (b) as estruturais, e (c) as de execução. Três atributos de modelo com tipos *booleano*, *inteiro* e *decimal* foram criados para exemplificar como os itens aparecem para serem manipulados pelo projetista.

Atributos

Nesta aba o projetista define quais atributos de células e de modelo farão parte do seu CA. Para cada atributo, deve ser escolhido um nome, tipo (booleano, inteiro ou decimal) e descrição. A interface foi pensada para suportar posteriores expansões para novos tipos (ver Figura 3.9).

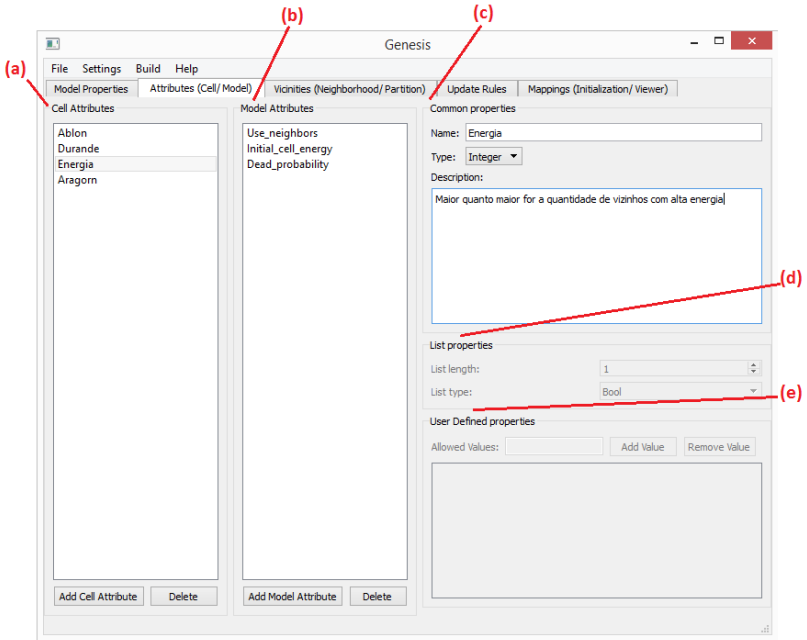


Figure 3.9: Aba de atributos. (a) Lista de atributos de células. (b) Lista de atributos de modelo. (c) Propriedades do atributo selecionado. (d) E (e) servirão a versões posteriores para suportar atributos do tipo lista (*list*) e arbitrários (*user defined* ou *tag*).

Vizinhanças

Esta aba permite a definição das vizinhanças que o projetista pretende utilizar nas regras de atualização. É exibida a lista de vizinhanças criadas e, ao selecionar uma delas, pode-se alterar seu nome, descrição e formato marcando que células relativas serão consideradas. (ver Figura 3.10). Foi reservado um espaço para a definição de partições (usadas em *block cellular automatas*) em posteriores versões do *genesis*. Além disso, a possibilidade de rotular vizinhos com nomes específicos, facilitando referências diretas, também está planejada em futuras versões.

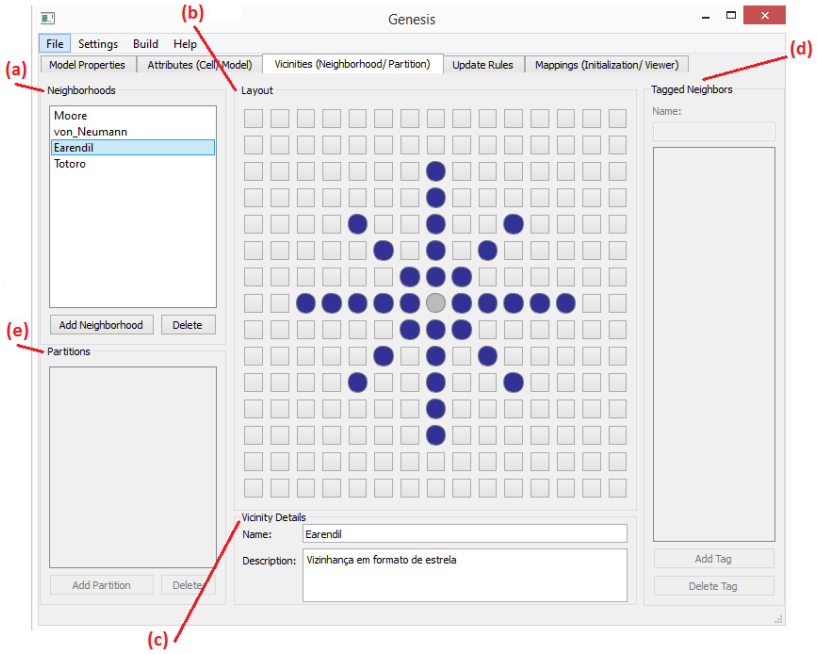


Figure 3.10: Aba de vizinhanças. (a) Lista de vizinhanças definidas. (b) Matriz de botões clicáveis, representando as células em torno de um referencial, para que seja possível definir as células contidas na vizinhança. (c) Informações básicas. (d) E (e) servirão a versões futuras, suportando o rótulo de vizinhos e definição de partições respectivamente.

Regras de Atualização

Nesta aba existe apenas um botão, o qual abre a janela do editor da linguagem visual. Aqui, o usuário poderá definir o comportamento das células a cada geração, bem como os efeitos dos mapeamentos de entrada (inicializações) e o cálculo das cores para os mapeamentos de saída (visualizações) (ver Figura 3.11). Os detalhes da linguagem visual serão dados na próxima seção.

Mapeamentos de Cor

Assim como na aba de atributos, esta aba oferece duas listas: mapeamentos de entrada (*color-attribute*) e mapeamentos de saída (*attribute-color*). O primeiro, define a lista de opções de inicialização/interação que serão permitidas no modelo, recebendo uma cor como entrada[1].

[1] Os mapeamentos são definidos a nível de células, então passar uma imagem como entrada ao modelo significa invocar um mapeamento para cada célula, lendo o respectivo pixel da imagem

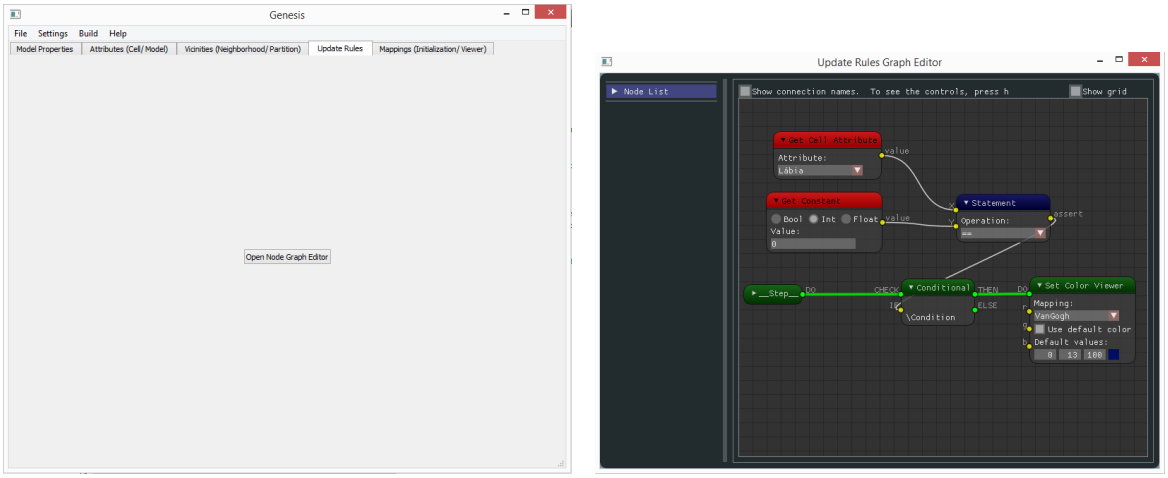


Figure 3.11: Telas referentes às *Regras de Atualização*. Na esquerda, a respectiva aba do *Genesis*, com o botão para abrir o editor da linguagem visual. Na direita, a janela do editor.

O segundo define as diferentes formas de apresentação em cores de uma dada configuração de células (ver Figura 3.12). É aconselhável preencher os campos de descrição, detalhando o efeito dos mapeamento de entrada, bem como o significado das cores nos mapeamento de saída, para que não haja má interpretação do comportamento observado na simulação.

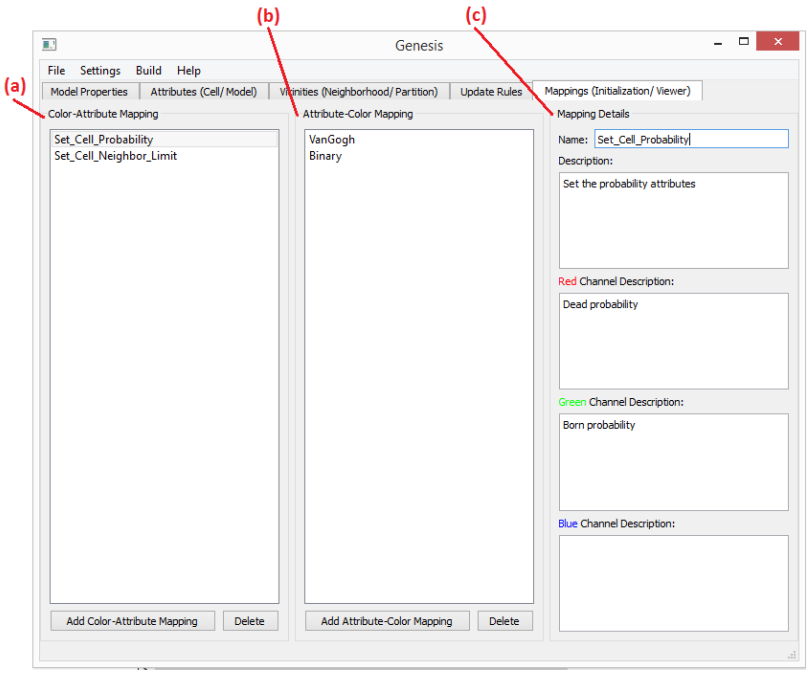


Figure 3.12: Aba de Mapeamentos. (a) Lista de mapeamentos de entrada/inicializações (*color-attribute*). (b) Lista de mapeamentos de saída/visualizações (*attribute-color*). (c) Informações básicas do mapeamento selecionado.

Simulator

O *simulator* é uma janela independente que executa um modelo gerado, e pode ser acessado através da opção *Run...* e *Generate Standalone Viewer* do item *Build* no menu principal.

Sua interface é composta pela tela principal de exibição da simulação e opções de configuração. Esta última sendo dividia em propriedades de execução e visualização do lado esquerdo, e opções de manipulação nos dados do modelo no lado direito (ver Figura 3.13).

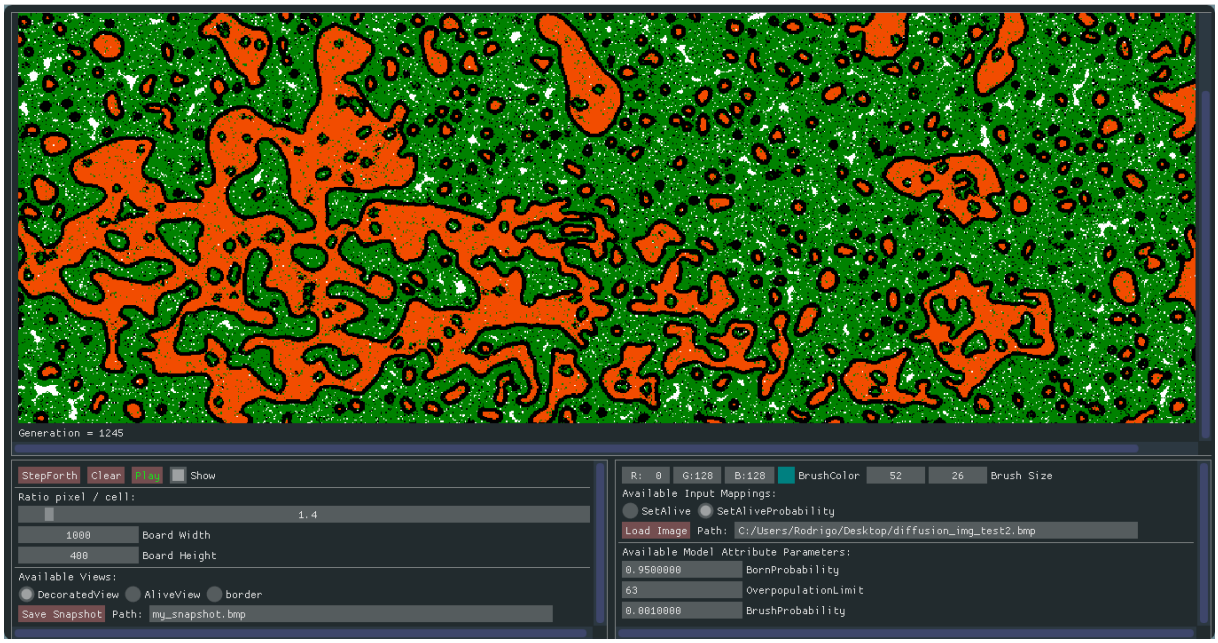


Figure 3.13: Janela do *Simulator* de um CA arbitrário. Acima a tela de simulação, e abaixo as diversas opções de configuração fornecidas ao usuário.

Além das opções tradicionais dos simuladores, o *Simulator* desenvolvido suporta as diferentes formas de visualização que foram criadas no modelo, referentes aos mapeamentos de saída (ver Figura 3.14), bem como os diferentes modos de inicialização que definem como o carregamento de imagens e cliques com o mouse serão interpretados pelas células. Outro ponto a se destacar é a exposição dos atributos de modelo ao usuário, conferindo-lhe o livre controle dos parâmetros durante a execução. O que facilita a investigação das propriedades do CA.

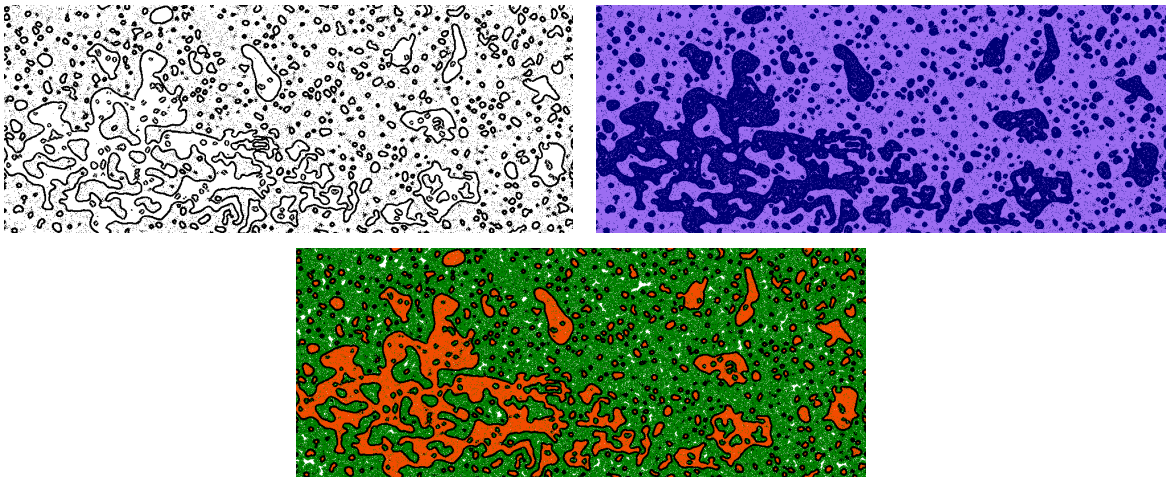


Figure 3.14: Três visualizações de um mesmo CA em execução. A primeira destaca os contornos das estruturas que se formaram, a segunda destaca as regiões, a terceira é enriquecida com cores mais apresentáveis. O projetista tem liberdade para definir quantas visualizações desejar.

3.4 Linguagem Visual

A linguagem visual desenvolvida se baseia em componentes conectáveis usados para criar um grafo de eventos e operações. Tal abordagem é utilizada por programas influentes e.g. as *blueprints* da *Unreal* [2], e os *composite nodes* do *Blender* [3] (ver Figura 3.15).

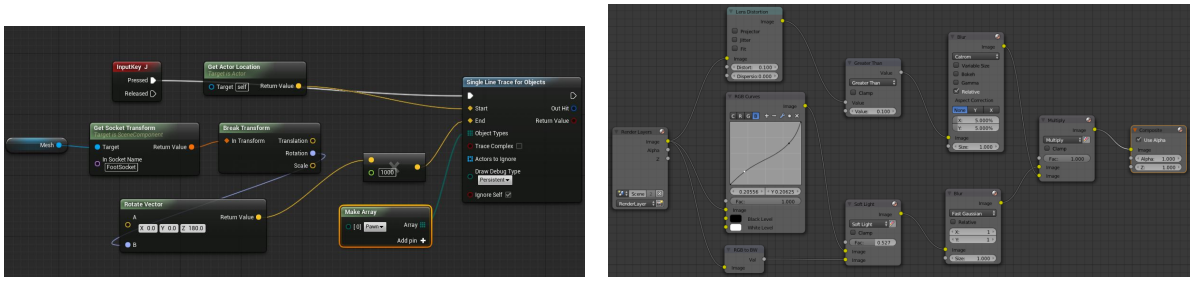


Figure 3.15: Exemplos de editor de nós. *Blueprints*, na esquerda, permitem que mesmo os *designers* de uma equipe, sem conhecimento de linguagens de programação, possam definir lógicas e eventos em um jogo. *Composite nodes*, na direita, são utilizados dentre outras coisas na pós-produção de imagens renderizadas.

De modo a simplificar a leitura dos grafos, o *Genesis* separa os nós em quatro grupos com finalidades distintas. São estes:

- *Controle* - Os nós deste grupo determinam o fluxo de chamada dos eventos, e são verdes. Similar ao *fluxo de controle* em linguagens de programação imperativas. De modo simplificado, o grafo composto apenas pelos nós de *controle*, formam o fluxograma do CA. Isto permite maior desacoplamento entre a manipulação de dados e o comportamento alto-nível desejado (ver Figura 3.16).

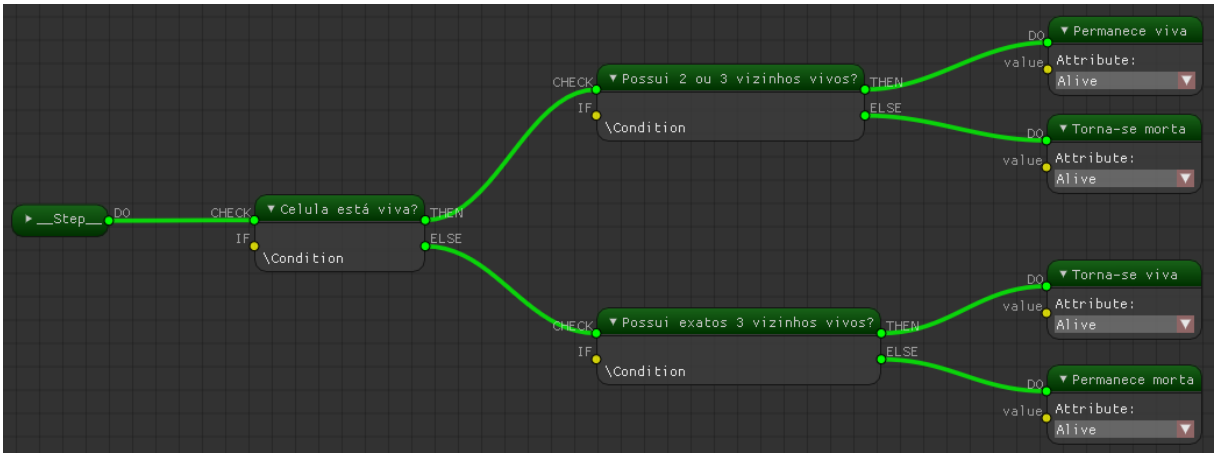


Figure 3.16: Grafo apenas com os nós de controle do CA *Game of Life* no *Genesis*.

[2] *Unreal* é uma *engine* de jogos desenvolvida pela *Epic Games*, e foi utilizada em títulos importantes como *BioShock*, *Mirror's Edge* e *Borderlands*.

[3] Blender é um programa de código aberto desenvolvido pela *Blender Foundation* de modelagem, animação, texturização, composição, renderização, edição de vídeo e criação de aplicações interativas como jogos.

- *Lógica* - Nós deste grupo fornecem a validação de enunciados lógicos e são azuis. Sempre produzem uma saída booleana, então em geral são utilizados para determinar condições de eventos. Por exemplo, checar se existe algum vizinho com um determinado valor num atributo específico.

- *Dados* - Todos os nós que não recebem entradas mas produzem dados de saída estão neste grupo, e são vermelhos. Incluindo nós que produzem constantes, números aleatórios e que leem dados das células.

- *Operações* - Os nós que recebem dados transformam-os e produzem novos dados de saída estão neste grupo, e são amarelos. Operações aritméticas, lógicas, de contagem em vizinhanças e agrupamento são contempladas por este grupo, fornecendo diversos tipos de cálculos que podem ser necessários ao projetista.

Definindo o fluxo de eventos e ações com os nós de *controle* e utilizando os nós de *lógica*, *dados* e *operações*, o modelo pode tornar-se tão complexo quanto necessário, e ainda manter-se acessível a leigos em programação (ver Figura 3.17). Cada nó possui *tooltips* que surgem ao posicionar o cursor sobre um determinado campo para informar o usuário, e ao selecionar um nó, é exibida uma explicação sobre seu comportamento (ver Figura 3.18)

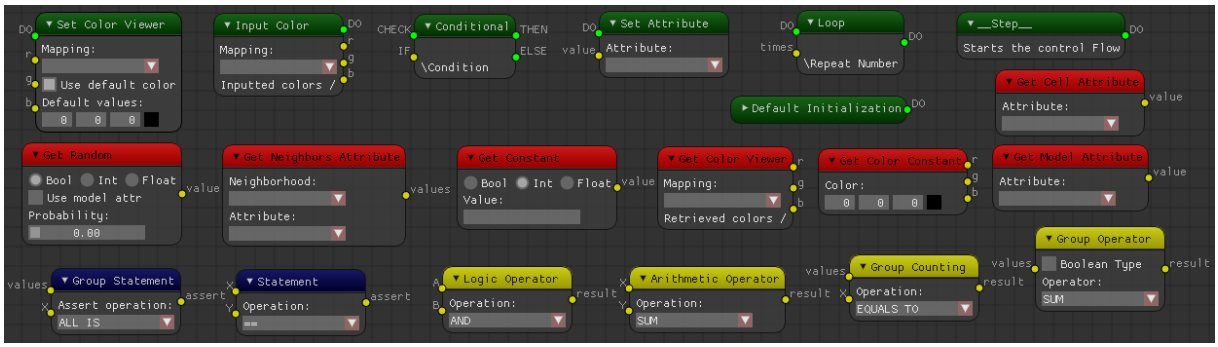


Figure 3.17: Todos os 20 nós disponíveis na versão atual do *Genesis*.

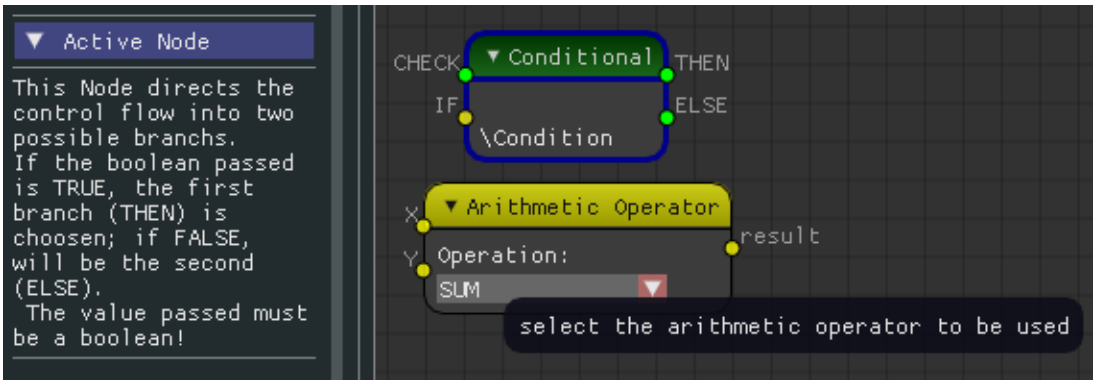


Figure 3.18: Informações adicionais sobre os nós exibidas no editor de grafos do *Genesis*. Após selecionar o nó *Conditional*, o campo *Active Node* exibe detalhes do seu comportamento. Ao posicionar o mouse sobre um campo do nó *Arithmetic Operator*, surge um *tooltip* de instrução.

3.5 Fluxo De Trabalho

Partindo do princípio de que o processo de modelagem de um autômato celular pode envolver vários ciclos de experimentação, é crucial que o projetista possa manipular todas as opções a qualquer momento. Além disso, existem vários casos de uso possíveis, dessa forma, o *Genesis* não estabelece uma sequencia de passos explícita. Seguem algumas possibilidades.

Investigar as propriedades de um modelo existente

Neste caso, o usuário se limitaria a usar e criar formas de visualizações distintas e alterar os atributos de modelo no *Simulator*, observando os efeitos. Para carregar configurações que julgar relevantes e registrar suas observações visuais, certamente utilizaria as opções de *inicialização por imagem* e *snapshot* da janela do simulador.

Criar um modelo com base em outro

Um usuário que parta de um modelo conhecido para alterar o modo como as células se comportam, se dedicará principalmente à aba *Update Rules*, uma vez que as configurações globais, e os atributos das célula e do modelo já estariam definidos. É provável que ele tenha a necessidade de criar diferentes formas de visualização para avaliar as alterações feitas no comportamento, então os mapeamentos também seriam manipulados.

Criar um modelo inteiramente novo

Se um modelo completamente novo está sendo desenvolvido, é natural que todas as opções do *Genesis* se façam necessárias. Uma possibilidade é: inicialmente, após um primeiro planejamento, o projetista definirá a estrutura básica do CA, com seus atributos, configurações globais e regras de atualização; posteriormente definirá meios de interagir e visualizar seu modelo; em seguida, diversos testes no simulador devem produzir *insights* sobre mudanças necessárias. Os passos podem se repetir tanto quanto necessário ao longo do processo experimental.

4

Implementação

Em seguida será apresentada uma visão geral da implementação do *Genesis*. O enfoque será a arquitetura, tecnologias e as abordagens técnicas utilizadas, que foram decisivas para suportar os requisitos da solução. Por não se tratar de um *guia de desenvolvimento*, detalhes práticos serão suprimidos.

4.1 Arquitetura

O *Genesis* é composto por três módulos implementados separadamente: modelo (*Model*), para reter todas as propriedades do CA, modelador (*Modeler*), que implementa a interface gráfica para manipular o modelo, e o simulador (*Simulator*), capaz de interpretar os modelos gerados e executá-los de forma interativa.

Modelo

O modelo compõe a classe que define qualquer CA construído no *Genesis*. Sua maior responsabilidade é oferecer uma interface para a manipulação de seus componentes, i.e. adicionar, remover e editar atributos, vizinhanças, e todas as outras propriedades (ver Figura 4.1). Também é de sua responsabilidade fornecer as funcionalidades de geração de código para a exportação.

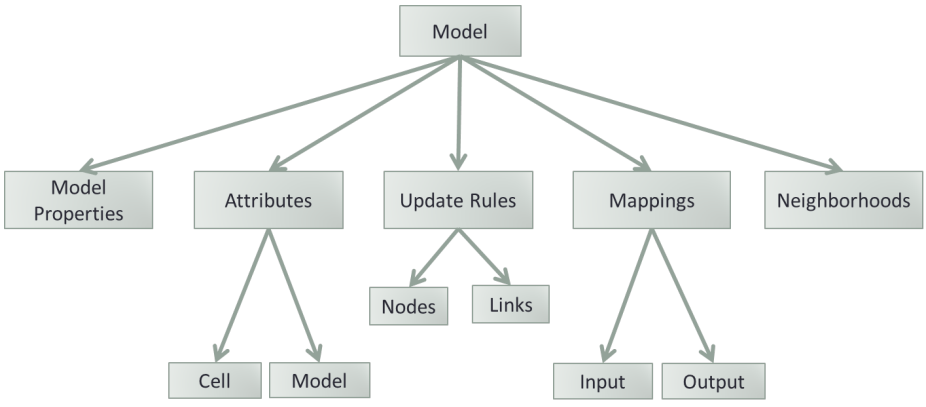


Figure 4.1: Esquema da arquitetura do *model*. As setas indicam a ordem de controle.

Modelador

Por se tratar de uma interface gráfica não trivial, fez-se necessário criar uma tela com sua própria implementação para cada tipo de configuração do modelo. Dessa forma, a janela principal se divide nas cinco abas já citadas, cada qual com sua implementação independente (ver Figura 4.2)). Tal modularidade simplifica possíveis expansões e aprimoramentos futuros, de forma que apenas as classes envolvidas com uma nova funcionalidade precisem ser alteradas.

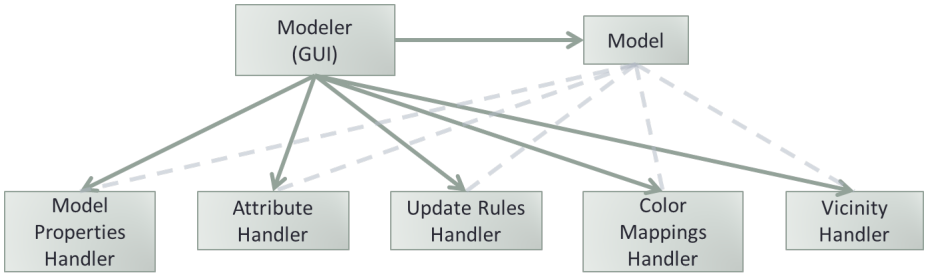


Figure 4.2: Esquema da arquitetura do *Modeler*. As setas indicam a ordem de controle. Deve-se observar que o *Modeler* controla o *Model*, e permite seu acesso a todas as abas, para que possam manipular as propriedades de sua responsabilidade.

A interface gráfica deve ser vista apenas como *um modo* de manipular o modelo, e portanto, não deve armazenar informações pertinentes à execução. Com essa desacoplação entre lógica e interação, novos modeladores podem ser implementados, por exemplo, para outras plataformas, sem precisar reimplementar o *Modelo* e *Simulator*.

Simulador

O *Simulator* não exige uma arquitetura rebuscada, uma vez que, a exibição e interação podem ser feitas em uma única tela. O único requisito arquitetural é ser capaz de interpretar qualquer CA no formato exportado pelo *Model*. E para isso, deve ser flexível para comportar os diferentes mapeamentos de entrada e saída, bem como atributos de modelo possíveis.

4.2 Tecnologias

O *Genesis* utiliza a linguagem *C++*, tanto na implementação dos três módulos descritos na seção anterior, quanto nos códigos gerados. Essa escolha foi fundamentada na portabilidade e eficiência, especialmente para o *Simulator*, para que seja possível executar CAs de grandes proporções sem maiores atrasos.

Como biblioteca gráfica para a criação das interfaces, foi utilizado majoritariamente o *Qt*, que é um ambiente multi-plataforma com suporte a geração de aplicações para os principais sistemas operacionais (i.e. *Linux*, *Windows* e *MAC*). Dessa forma, a janela do *Modeler* e suas abas, foram feitas em *Qt*. O ambiente de desenvolvimento integrado *Qt Creator* também foi utilizado na estruturação dos arquivos do projeto.

Adicionalmente, para o *Simulator* e a janela do editor de nós (nas regras de atualização), a biblioteca *Dear ImGui* precisou ser utilizada. Isto porque a *Dear Imgui* segue o paradigma *Imediate Mode GUI* em detrimento do *Retained Mode GUI* do *QT*. Isto significa que todos os elementos gráficos (como os nós da VPL) são desenhados a cada instante, facilitando por exemplo a alteração da forma do nó em resposta a seus valores (ver Figura 4.3).

Figure 4.3: Exemplo de variações de forma apresentadas pelo mesmo nó (*Get Random*), dependendo das opções selecionadas.

O *Visual Studio 2013* foi utilizado para organizar o projeto do *Simulator*, e seu compilador (*MSVC*) é invocado na exportação das aplicações independentes (*Simulator*). Ou seja, é um requisito do *Genesis*, que o usuário possua este compilador instalado em sua máquina. No entanto, tal requisito pode ser eliminado futuramente, ao incluir no pacote do *Genesis* alguma distribuição de compiladores C++ livres.

Para o versionamento dos códigos ao longo do desenvolvimento, foi utilizada a plataforma online *GitHub*. Um repositório[1] público foi criado, onde imagens e informações sobre o projeto podem ser encontradas. Neste repositório, futuramente podem ser adicionados exemplos de modelos, bem como tutoriais mais detalhados para guiar novos usuários.

4.3 Abordagens Técnicas

Além das decisões teóricas já abordadas em seções anteriores, e.g. delimitar o tipo de modelo de CA que será considerado no *Genesis*, ou escolher uma interface gráfica adequada; várias decisões práticas tiveram de ser tomadas, e influenciaram fortemente o curso do trabalho. Seguem algumas considerações técnicas que se mostraram relevantes durante o processo de implementação do *Genesis*.

[1] O repositório pode ser acessado em: *https://github.com/rff255/Genesis*

Editor de nós em janela separada

A razão pela qual o editor de nós não pôde ser embutido na aba *Update Rules*, está ligada a uma incompatibilidade entre o *Qt* (usado na aba) e a *Dear ImGui* (usada no editor). Isto acontece porque ambas as bibliotecas gráficas necessitam de uma referência de contexto para desenhar na tela, e o *Qt* possui seu próprio tipo de referência. Mesmo usando um *OpenGL Widget*[2] do *Qt*, não foi possível compatibilizá-los.

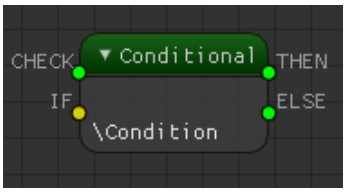
DLL gerada possui pouca utilidade

Originalmente o produto principal do *Genesis*, após a modelagem, seria uma biblioteca que pudesse tanto ser importada por um *Simulator*, quanto para uso geral (e.g. incluir em outros projetos). No entanto, observou-se que as *DLL*s não possuem mecanismos de portabilidade, de modo que até mesmo utilizando a mesma máquina e o mesmo compilador, podem haver erros decorrentes de diferenças nas opções de compilação utilizadas. Com isso, a *DLL* gerada só irá funcionar em códigos que utilizem o mesmo compilador, versão e opções de compilação.

Para contornar isto, optou-se por permitir também a exportação do código puro (*.h* e *.cpp*), para que pudesse ser incluído em outras aplicações, bem como o *Standalone Application* que compila o código gerado junto ao projeto do *Simulator*. Assim, por não ter dependência do *Genesis*, ser executável e estar pronta para visualização, a aplicação independente acabou sendo um recurso muito mais valioso para o compartilhamento de modelos.

Compilar a linguagem visual em um código C++

O processo de geração do código com base no grafo construído pelo usuário consiste num processo de compilação. Para tanto, na definição de cada nó existe uma função chamada *Eval* que retorna um texto com o código que aquele nó representa, já com os nomes das propriedades de modelo corretas. O processo inteiro é recursivo, de modo que um nó com portas de entrada, irá invocar a função *Eval* dos nós a ele conectados[3] e depois escrever seu próprio código, considerando que os dados de entrada estão definidos (pseudo-código na Figura 4.4).




```
1  # Invocar função Eval() do nó conectado na porta IF
2  if(IF_NODE_OUTPUT) {
3    # Invocar função Eval() do nó conectado na porta THEN
4  } else {
5    # Invocar função Eval() do nó conectado na porta ELSE
6  }
```

Figure 4.4: Exemplo do esquema usado na geração de código do nó *Conditional* em pseudo-código. Usando uma convenção de nomenclatura, é possível saber qual o nome da variável (na imagem ilustrado como *IF_NODE_OUTPUT*) que será criada na invocação do *Eval()* no nó conectado à porta *IF*. O código real pode ser visto no apêndice, na Figura 1.

[2]OpenGL é uma biblioteca gráfica livre que fornece uma interface aos diversos recursos de desenho do *hardware*

[3]Nem sempre os nós conectados são invocados. Foi necessário manter uma pilha extra para lidar com o escopo de cada nó, e evitar a redefinição de variáveis, para quando um nó tem sua saída conectada a várias entradas

5
Experimentos

Em seguida serão apresentados alguns experimentos feitos para validar o funcionamento da solução proposta. Serão apresentados três casos de uso, demonstrando a modelagem de CAs clássicos da literatura. Além disso, será feita uma análise inicial a respeito do desempenho de uma regra exportada pelo *Genesis*.

5.1 Casos de uso

Seguem abaixo os três casos de uso. *Game of Life*, descoberto por John Conway, *WireWorld*, descrito inicialmente por Brian Silverman e o CA elementar 1D *Rule110*. Para evitar repetições, será descrito o passo a passo da criação no primeiro experimento e apenas o essencial para os dois últimos.

Game of Life

Game of Life (ou *GoL*) é um CA 2D, turing-completo, em cujas células possuem apenas duas configurações possíveis (*viva* e *morta*). No *GoL*, é possível identificar diversas estruturas, como osciladores, naves e geradores de outras estruturas. Sua regra se resume a quatro pontos:

1. Qualquer célula *viva* com menos de dois vizinhos *vivos* morre de solidão.
2. Qualquer célula *viva* com mais de três vizinhos vivos morre de superpopulação.
3. Qualquer célula *morta* com exatamente três vizinhos vivos se torna uma célula *viva*.
4. Qualquer célula *viva* com dois ou três vizinhos vivos permanece *viva*.

Para implementar o *Gol*, um usuário deve (não necessariamente nesta ordem) realizar os seguintes procedimentos:

- Na aba *Model Properties*, definir as propriedades do modelo como desejar. Exemplo: nome, descrição e o comportamento da fronteira

- Na aba *Attributes*, criar um atributo para armazenar o estado *vivo/morto*. Naturalmente um atributo booleano com nome *alive* é uma opção sugestiva.

- Na aba *Vicinities*, definir a vizinhança. No caso do *GoL*, a vizinhança utilizada é a de *Moore*, como ilustrada na Figura 5.1.

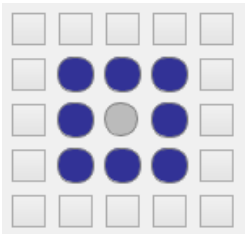


Figure 5.1: Vizinhança de *Moore*. A bola cinza representa a célula central, e as azuis são células da vizinhança. Os quadrados são células não contempladas por esta vizinhança.

- Como todo CA precisa de uma forma de visualização e uma forma de interagir para alterar as informações das células, deve-se criar ao menos um mapeamento de entrada e outro de saída na aba *Mappings*. Dado que são apenas dois estados, a visualização mais simples seria o binário preto e branco, e uma possível forma de inicialização por cor seria: se o canal vermelho for diferente de zero, torne a célula viva.

- Por fim, deve-se definir as regras de atualização na aba *Update Rules*. Aqui, o comportamento das células a cada geração e os mapeamentos devem ser definidos utilizando a linguagem visual. O resultado pode ser visto na Figura 5.2.

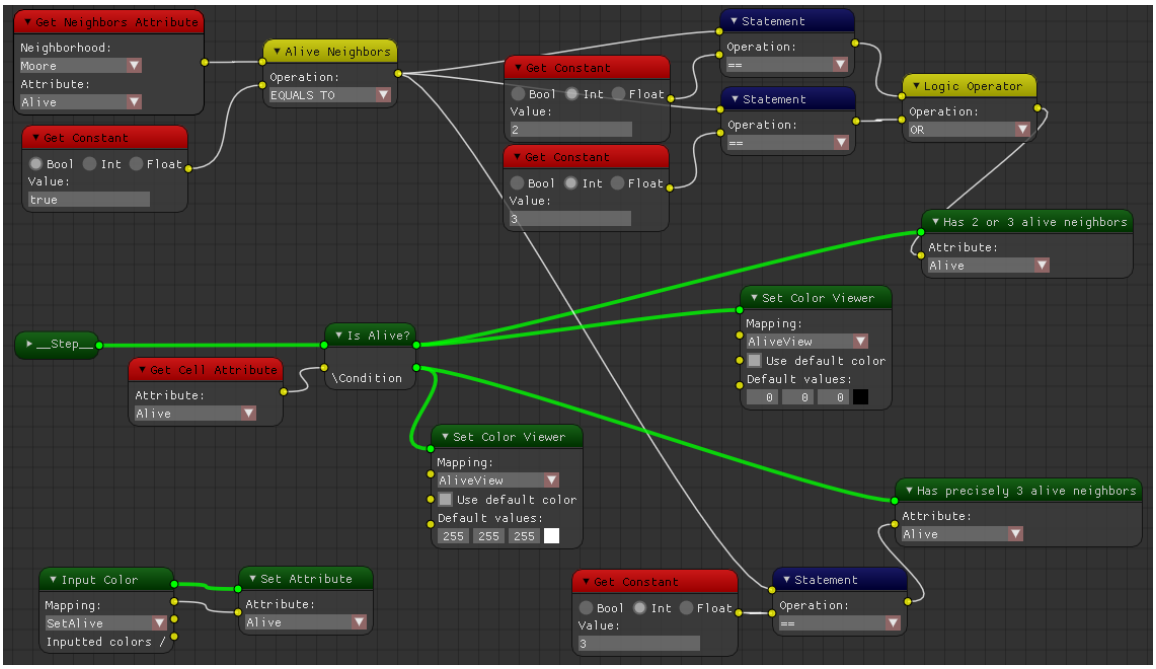


Figure 5.2: Regras de atualização no exemplo do *Game of Life*. Pode-se observar que a leitura dos nós de controle (em verde) determinam a lógica principal, tal qual um fluxograma. Os mapeamentos de entrada e saída usados foram *SetAlive* e *AliveView*, respectivamente.

- Com tudo pronto, o usuário pode executar ou exportar o modelo, na opção *Build* do menu principal. O resultado pode ser visto na Figura 5.3.

Figure 5.3: Simulação do modelo *GoL* criado.

Para ilustrar melhor as capacidades do *Genesis*, alteraremos um pouco o modelo acima, incluindo:

- Um mapeamento de entrada extra (*SetAlive_Probabilistic*), que permite o uso de um parâmetro de probabilidade para definir que células vão nascer, que pode ser útil para avaliar o comportamento do CA partindo de condições iniciais aleatórias. Para tanto, deve-se criar um atributo de modelo, que ficará visível no *Simulator* (ver Figura 5.4).

- Um mapeamento de saída extra (*DecoratedView*), para tornar a visualização mais amigável, bem como destacar padrões diferentes (ver Figura 5.4). Este mapeamento usa a quantidade de vizinhos vivos para determinar o valor do componente vermelho.

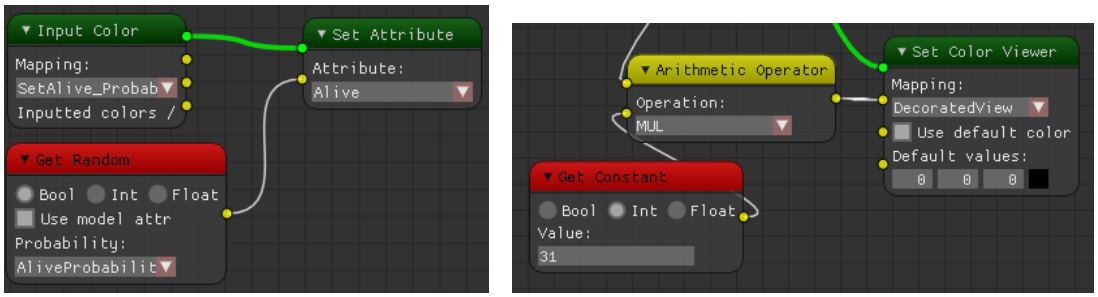


Figure 5.4: Definição dos novos mapeamentos. Na esquerda, o novo mapeamento de entrada *SetAlive_Probabilistic*, na direita o novo mapeamento de saída *DecoratedView*. O fio branco da parte superior representa o número de vizinhos vivos.

- Substituir as constantes da regra *GoL* por atributos de modelo i.e. o número de vizinhos vivos mínimo passa a ser um parâmetro *UnderPopulation*, o máximo *OverPopulation* e a quantidade em que uma célula nasce *BornNumber*. A Figura 5.5 ilustra que a alteração para realizar isto é mínima.

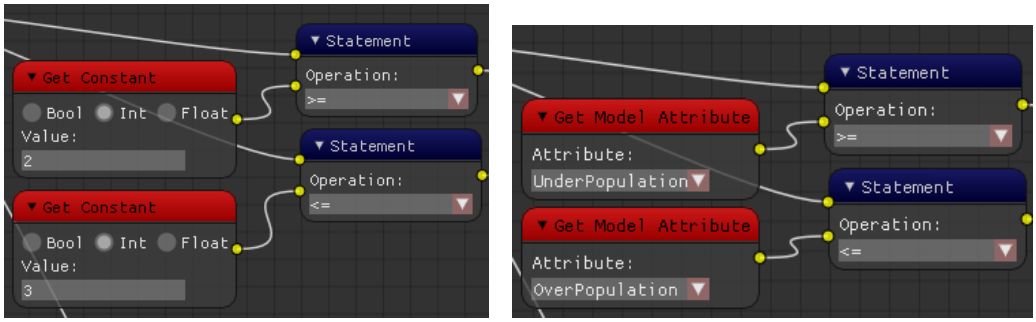


Figure 5.5: Substituir os nós *Get Constant* pelos respectivos *Get Model Attribute* são as únicas alterações necessárias.

- Adicionar duas vizinhanças extras: *Castiel* e *Coagulation* (Ver Figura 5.6). Não foi implementado um mecanismo automático para a troca de vizinhanças em tempo de simulação, mas é possível fazê-lo. Para isso, adicionou-se um atributo de célula *AliveNeighbors* que vai armazenar o número de vizinhos vivos de acordo com a vizinhança indicada no atributo de modelo *Neighborhood*. Isto significa que na regra de atualização, deve ser checada qual a vizinhança selecionada no atributo *Neighborhood*, e definir o valor de *AliveNeighbors* de acordo. Por fim, nos locais do grafo onde era calculada a quantidade de vizinhos, deve-se usar o atributo de célula *AliveNeighbors* (Ver Figura 2).

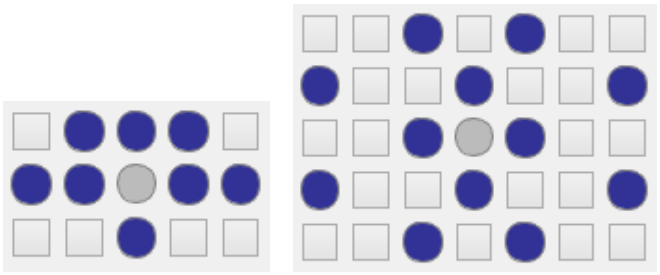


Figure 5.6: Vizinhanças extras adicionadas. Vizinhança *Castiel* e *Coagulation*, respectivamente.

Com isso, o modelo torna-se muito mais dinâmico, no sentido de possibilitar mais experimentações e análises. No *Simulator* a interação torna-se mais completa, uma vez que todos os parâmetros criados na forma de atributos de modelo podem ser manipulados, bem como o uso da inicialização com probabilidade randômica. Além disso, a visualização diferente é capaz de ressaltar padrões durante a execução (ver Figura 5.7). A simulação com a vizinhança *Castiel*, bem como a comparação entre as visualizações no *GoL* podem ser vistas na Figura 3 e 4.

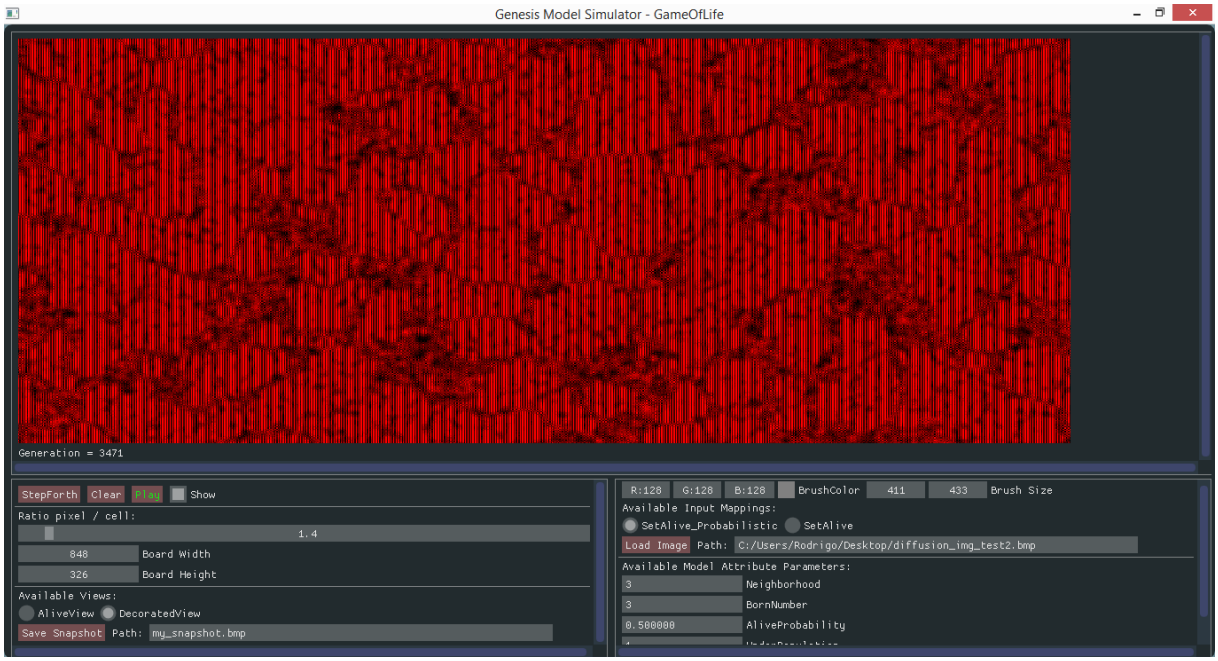


Figure 5.7: *Simulator* depois das alterações no modelo. Em execução a vizinhança *Coagulation*, utilizando a visualização *DecoratedView*. Este CA exibe padrões que lembram vasos sanguíneos e a coagulação. A única diferença para o *GoL*, além da vizinhança, é o parâmetro *UnderPopulation* ser 1 em vez de 2.

WireWorld

Wireworld é um CA 2D, turing-completo, em cujas células possuem quatro configurações possíveis (*vazia*, *fio*, *cabeça* e *calda*). O *Wireworld* é capaz de simular circuitos complexos, uma vez suporta estruturas com comportamento similar às portas lógicas. As regras deste CA são:

1. Uma célula *vazia* sempre continuará *vazia*.
2. Uma célula *cabeça* sempre se tornará *calda*.
3. Uma célula *cauda* sempre se tornará *fio*.
4. Uma célula *fio* se tornará *cabeça* se houver uma ou dois vizinhos no estado *cabeça*. Senão permanece *fio*.

Para implementar o *Wireworld*, é necessário apenas criar um atributo de célula inteiro, para armazenar um dentre os quatro possíveis estados, e definir um mapeamento de entrada e outro de saída. A vizinhança utilizada é a de *Moore* (da Figura 5.1). Uma possível implementação das regras de atualização deste CA pode ser a ilustrada na Figura 5.8, e o resultado da execução pode ser visto na Figura 5.9.

Rule110

Rule110 é um CA elementar 1D extremamente simples, porém também foi provado como turing-completo. Um CA elementar possui apenas dois estados (*0* e *1*, ou *vivo* e *morto*)

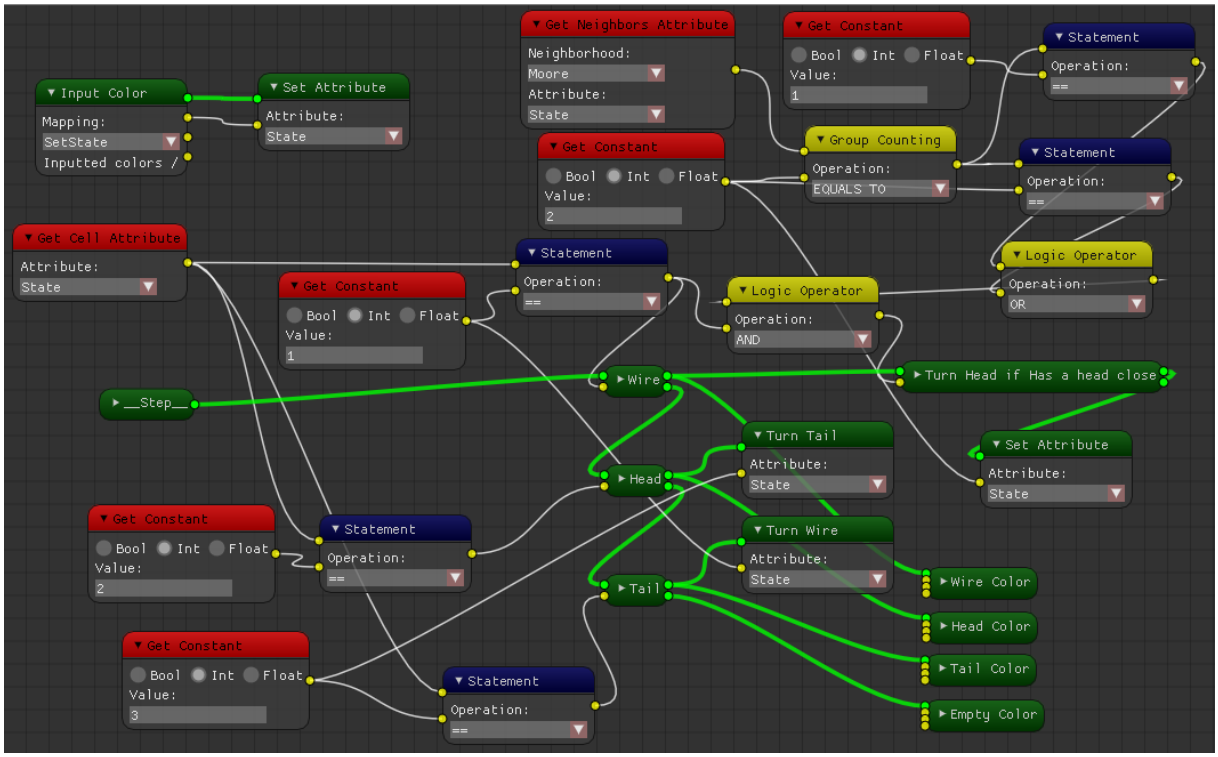


Figure 5.8: Uma das possíveis formas de se implementar as regras de atualização do CA *Wireworld*. É relevante observar que não apenas a disposição dos nós é arbitrária, como também exitem vários meios de se definir a mesma regra, seja em prol da legibilidade, ou da praticidade.

Figure 5.9: *Simulator* executando o CA *Wireworld*. O mapeamento de entrada *SetAlive* utiliza o valor da componente vermelha (0 na imagem) para definir o estado da célula estimulada.

e a vizinhança é composta pelos dois vizinhos imediatos apenas. Suas regras de transição são comumente escritas em forma de tabela (ver Figura 5.10), onde na primeira linha são sequenciadas todas as configurações possíveis da célula (número central) com os dois vizinhos e a segunda linha indica para qual configuração respectiva a célula irá.

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

Figure 5.10: Tabela de transição do CA elementar *Rule110*. Uma observação útil é a de que a célula só se torna *morta* em três situações: quando ela e suas vizinhas estão vivas; quando ela e suas vizinhas estão mortas; e quando apenas a vizinha esquerda está viva.

Este CA foi escolhido para demonstrar que, embora o *Genesis* tenha sido idealizado principalmente para modelar CAs 2D, é possível definir CAs 1D também. Isto se deve ao fato de que a definição das vizinhanças é completamente livre, então a diferença é que foram utilizadas duas vizinhanças peculiares para implementá-lo (ver Figura 5.11).

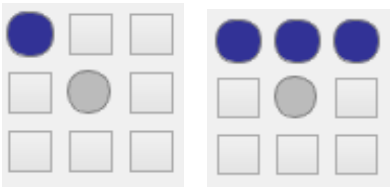


Figure 5.11: Vizinhanças utilizadas para implementar o CA *Rule110*. Na esquerda, a vizinhança *left*, e na direita a *ThreeUp*

Além da implementação mais simples, foi adicionado um mapeamento extra de entrada (similar ao supracitado *SetAlive_probabilistic* adicionado ao *GoL*), e dois modos de visualização extra: *CategorizedView*, que separa as três diferentes formas de uma célula ir para o estado *morto* (amarelo, vermelho e verde) e *DensityMap* que utiliza duas vizinhanças extras (ver Figura 5.12) para criar uma visualização que destaque padrões diferentes na simulação. O grafo da regra de atualização sem as visualizações e interações pode ser visto na Figura 5.13, e imagens da simulação com as três visualizações na Figura 5.14.

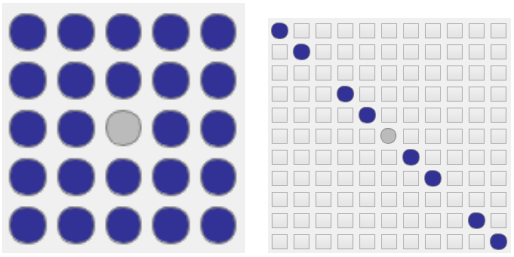


Figure 5.12: Vizinhanças adicionais usadas para criar a visualização *DensityMap* do CA *Rule110*. Estes formatos foram escolhidos após uma observação sobre as estruturas comuns deste autômato.

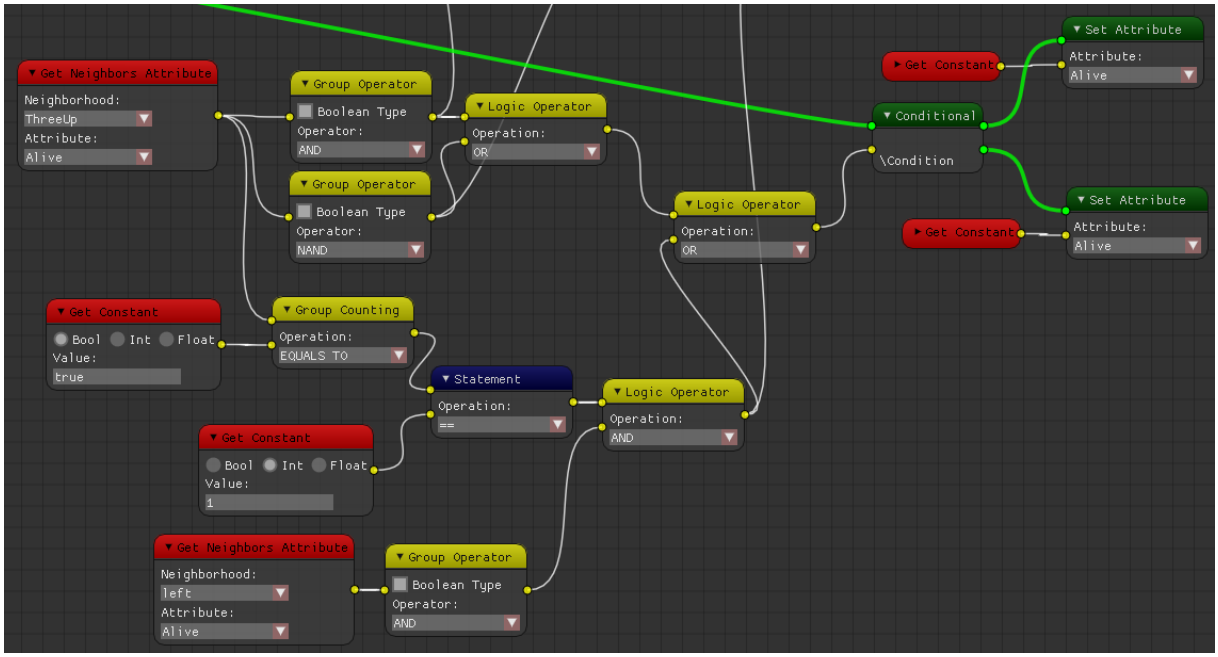


Figure 5.13: Uma possível implementação do CA *Rule110*. Os nós relativos aos mapeamentos de entrada e saída foram suprimidos afim de expor apenas o essencial.

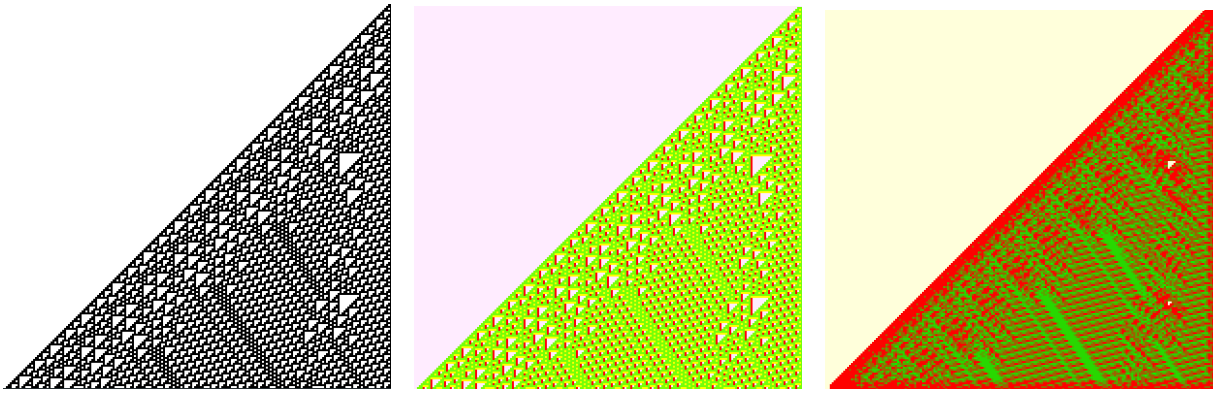


Figure 5.14: Três visualizações (*AliveView*, *CategorizedView* e *DensityView*) da execução do autômato celular *Rule110*.

5.2 Desempenho

Uma vez que o objetivo deste trabalho não está relacionado a uma melhora ou investigação de desempenho, mas sim na capacidade de modelagem dos programas, foi feita apenas uma análise inicial do tempo de execução de um CA exportado pelo *Genesis* sem grande precisão ou acurácia. Para tanto, foi utilizado o próprio *Simulator*, com o modelo final do primeiro experimento da seção anterior (*GoL* parametrizável, com vizinhanças e visualizações extras).

Em um universo de largura 500 e altura 500 (250.000 células) inicializado randomicamente, foram necessários 14,211 segundos para iterar 500 gerações[1] (imagem do experimento na Figura 5.15). Isto significa uma média de 28 milissegundos/geração. Ou: 0,11 microssegundos/geração para cada célula. É sabido que: quanto maior for o custo computacional da regra, maior será o tempo necessário, então esta medida não deve ser usada como constante. Porém este teste serve para indicar a ordem de grandeza do custo intrínseco dos códigos exportados.

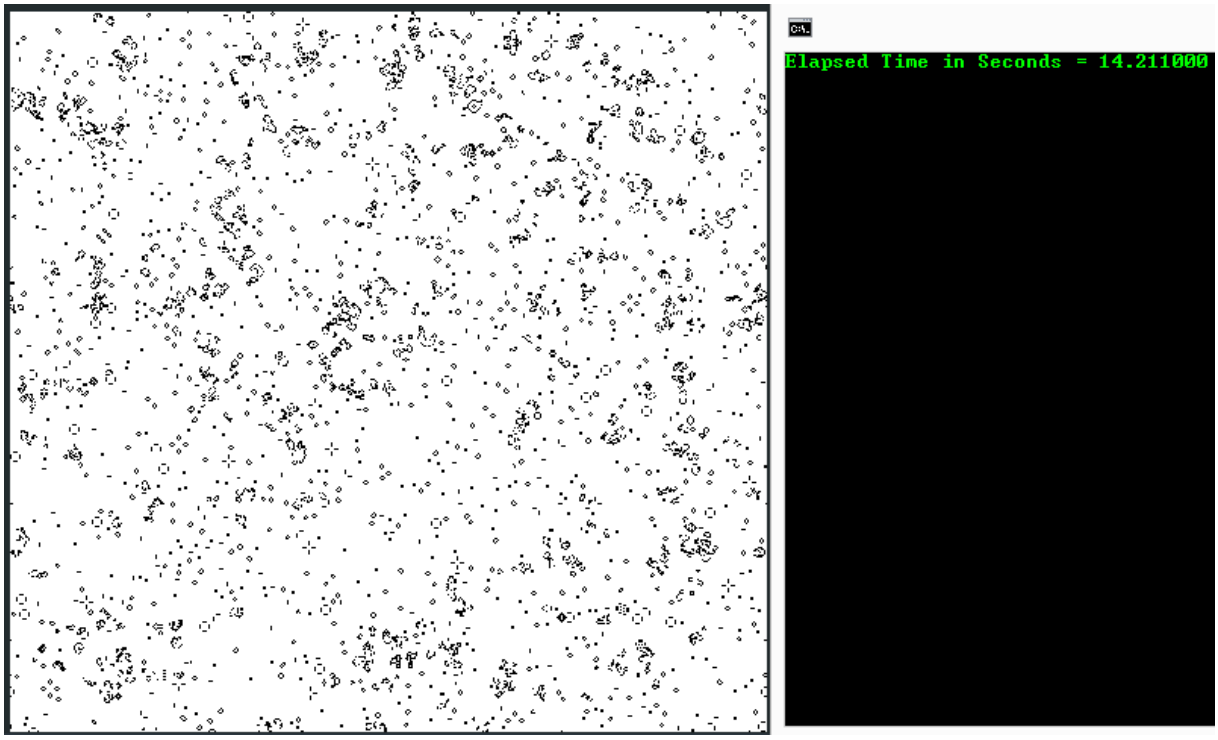


Figure 5.15: Imagem do experimento feito para mensurar o desempenho do código de um modelo exportado. O modelo executado foi o *GoL* com as adições apresentadas no primeiro caso de uso da seção anterior. O universo usado possui dimensão 500 por 500, e foram iteradas 500 gerações.

A simulação em todos os modelos executados manteve-se fluida, e isto se deve ao fato de que o modelo em si é completamente independente da interface gráfica, armazenando e processando apenas o indispensável à execução. Além disso o uso da linguagem *C++*, que por um lado dificultou o processo de exportação e compilação, foi uma decisão crucial para a obtenção deste desempenho.

[1] O computador utilizado possui processador i5-3210M CPU 2,50GHz, memória 6GB e Sistema operacional Windows 8.1 64bits

6

Conclusão

Autômatos Celulares compõem uma classe de modelos aplicáveis aos mais distintos propósitos, contemplando diversos campos da ciência e industria. No entanto, não existe uma plataforma bem estabelecida que ofereça um suporte completo e acessível para o desenvolvimento de novos modelos. Este trabalho propôs uma solução a este problema, fundamentada no desenvolvimento de uma plataforma integrada que utiliza uma linguagem de programação visual.

6.1 Considerações

Foi feita uma avaliação breve das diversas soluções encontradas, por meio da qual foi possível identificar características prejudiciais e funcionalidades desejáveis que guiaram o desenvolvimento do *Genesis*. Como resultado, o programa implementado possui capacidade de modelar CA de modo irrestrito, oferecendo ainda funcionalidades extras à teoria, como o suporte à criação de diferentes modos de visualização e interação, além da definição de parâmetros.

Em todo o processo de modelagem utilizando o *Genesis*, não é requerido que o usuário digite sequer uma linha de código. Isto torna o programa acessível mesmo a leigos em programação, permitindo que cientistas de outras áreas possam realizar suas próprias experiências, sem depender de terceiros ou de qualificações prévias específicas.

Além do modelador de CA, um módulo de simulação (*Simulator*) foi desenvolvido para servir de interface às capacidades dos modelos desenvolvidos no *Genesis*. Por meio dele, um projetista pode interagir com tudo o que foi definido, e ainda compartilhar aplicações independentes (executáveis) do seu modelo com pessoas que não possuam o *Genesis* mas tenham interesse em visualizar os resultados.

Os experimentos realizados ilustram algumas das capacidades do *Genesis*, ao modelar três CAs amplamente edostudados da literatura. Além disso, foi realizado um teste inicial de tempo de execução, para assegurar que o desempenho não foi sacrificado em prol das outras funcionalidades.

6.2 Trabalhos Futuros

Testes de usabilidade

Por se tratar de uma ferramenta de suporte a criação, que oferece uma interface direta ao usuário, é fundamental que sejam realizados testes de usabilidade. Isto servirá para elucidar pontos fracos do programa, bem como esclarecer a direção que o desenvolvimento deve seguir em versões futuras.

Reformulação do conjunto de nós na linguagem visual

Durante a concepção da ideia, e desenvolvimento, não é possível ter uma noção clara dos nós que serão mais utilizados, bem como a melhor forma de fazê-los. Então, ao fim do trabalho foi possível observar que alguns nós raramente foram utilizados, e que alguns conjuntos de nós comumente se repetiam. Portanto, é importante realizar uma análise crítica para remodelar, remover e adicionar o que se mostrar necessário, mantendo a linguagem visual o mais concisa possível, de modo que o projetista não perca o foco.

Melhorar interação no editor da linguagem visual

Assim como quais nós estão a disposição, o modo como posso manipulá-los influenciará a facilidade que o usuário terá em se adaptar e utilizar com eficiência o editor. Diversas funcionalidades mostraram-se desejáveis, dentre elas:

- Possibilidade de agrupar nós, para transladar, esconder e deletar todos de uma vez.
- Possibilidade de criar um nó a partir de um grupo de nós, de modo que um cálculo ou lógica complexos que seja realizado em um conjunto de nós pudessem ser "fundidos" para posterior uso.
- Documentação mais detalhada do uso e funcionamento de cada nó, de preferência incluindo *tooltips* em cada porta, para guiar o usuário.
- Permitir a adição de nós por meio de uma caixa de texto de filtragem, para agilizar a busca pelo nó desejado.
- Alertas de tipos incompatíveis nas conexões e de uso incorreto dos nós.

Melhorar interação no *Simulator*

Embora o *Simulator* não tenha sido o foco deste trabalho, ao decorrer do projeto, ele mostrou-se extremamente importante. Dessa forma, convêm aprimorá-lo para permitir mais opções de interação e.g. definir uma célula modelo em tempo de execução e utilizá-la como *brush*; e exibir as descrições escritas pelo usuário nos atributos, mapeamentos vizinhanças.

Referências

AMLANI, I. et al. Digital logic gate using quantum-dot cellular automata. **science**, [S.l.], v.284, n.5412, p.289–291, 1999.

AVOLIO, M. et al. An extended notion of Cellular Automata for surface flows modelling. **WSEAS Transactions on Computers**, [S.l.], v.2, n.4, p.1080–1085, 2003.

CHANG, C.-L.; ZHANG, Y.-J.; GDONG, Y.-Y. Cellular automata for edge detection of images. In: MACHINE LEARNING AND CYBERNETICS, 2004. PROCEEDINGS OF 2004 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2004. v.6, p.3830–3834.

DE CASTRO, L. N. **Fundamentals of natural computing**: basic concepts, algorithms, and applications. [S.l.]: CRC Press, 2006.

ESLAMI, Z.; RAZZAGHI, S.; AHMADABADI, J. Z. Secret image sharing based on cellular automata and steganography. **Pattern Recognition**, [S.l.], v.43, n.1, p.397–404, 2010.

FREIWALD, U.; WEIMAR, J. R. JCASim—a Java system for simulating cellular automata. In: **Theory and Practical Issues on Cellular Automata**. [S.l.]: Springer, 2001. p.47–54.

HAMAMCI, A. et al. Cellular automata segmentation of brain tumors on post contrast MR images. **Medical Image Computing and Computer-Assisted Intervention–MICCAI 2010**, [S.l.], p.137–146, 2010.

ILACHINSKI, A. **Cellular automata**: a discrete universe. [S.l.]: World Scientific Publishing Co Inc, 2001.

JOHNSON, L.; YANNAKAKIS, G. N.; TOGELIUS, J. Cellular automata for real-time generation of infinite cave levels. In: WORKSHOP ON PROCEDURAL CONTENT GENERATION IN GAMES, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.10.

KLOPFER, E. et al. The simulation cycle: combining games, simulations, engineering and science using starlogo tng. **E-Learning and Digital Media**, [S.l.], v.6, n.1, p.71–96, 2009.

LAFE, O. Data compression and encryption using cellular automata transforms. **Engineering Applications of Artificial Intelligence**, [S.l.], v.10, n.6, p.581–591, 1997.

LIANG, J. **Simulating crimes and crime patterns using cellular automata and GIS**. 2001. Tese (Doutorado em Ciência da Computação) — University of Cincinnati.

MARTIN, R. C. **Clean code**: a handbook of agile software craftsmanship. [S.l.]: Pearson Education, 2009.

MELLO, R. F. L.; CASTILHO, C. A Stuctured Discrete Model for Dengue Fever Infections and the Determination of R_0 from Age-Stratified Serological Data. **Bulletin of mathematical biology**, [S.l.], v.76, n.6, p.1288–1305, 2014.

MIRANDA, E. R. Evolving cellular automata music: from sound synthesis to composition. In: WORKSHOP ON ARTIFICIAL LIFE MODELS FOR MUSICAL APPLICATIONS, 2001. **Proceedings...** [S.l.: s.n.], 2001.

QIU, S. et al. Genome evolution by matrix algorithms: cellular automata approach to population genetics. **Genome biology and evolution**, [S.l.], v.6, n.4, p.988–999, 2014.

RAABE, D.; BECKER, R. C. Coupling of a crystal plasticity finite-element model with a probabilistic cellular automaton for simulating primary static recrystallization in aluminium. **Modelling and Simulation in Materials Science and Engineering**, [S.l.], v.8, n.4, p.445, 2000.

SELVAPETER, P. J.; HORDIJK, W. Cellular automata for image noise filtering. In: NATURE & BIOLOGICALLY INSPIRED COMPUTING, 2009. NABIC 2009. WORLD CONGRESS ON. **Anais...** [S.l.: s.n.], 2009. p.193–197.

TISUE, S.; WILENSKY, U. Netlogo: a simple environment for modeling complexity. In: INTERNATIONAL CONFERENCE ON COMPLEX SYSTEMS. **Anais...** [S.l.: s.n.], 2004. v.21, p.16–21.

TRALIC, D. et al. Detection of duplicated image regions using cellular automata. In: SYSTEMS, SIGNALS AND IMAGE PROCESSING (IWSSIP), 2014 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2014. p.167–170.

WHITE, R.; ENGELEN, G. Cellular automata and fractal urban form: a cellular modelling approach to the evolution of urban land-use patterns. **Environment and planning A**, [S.l.], v.25, n.8, p.1175–1199, 1993.

WOLF-GLADROW, D. A. **Lattice-gas cellular automata and lattice Boltzmann models**: an introduction. [S.l.]: Springer, 2004.

WOLFRAM, S. Cryptography with cellular automata. In: CONFERENCE ON THE THEORY AND APPLICATION OF CRYPTOGRAPHIC TECHNIQUES. **Anais...** [S.l.: s.n.], 1985. p.429–432.

WOOTTON, J. T. Local interactions predict large-scale pattern in empirically derived cellular automata. **Nature**, [S.l.], v.413, n.6858, p.841–844, 2001.

WUENSCHE, A. Discrete dynamics lab (ddlab). **Software and Manual**, [S.l.], 1996.

YATAPANAGE, N. Cellular Automata as a Model for Dynamic Leaf Structure., [S.l.], 2003.

ZHANG, Q. et al. Simple cellular automaton-based simulation of ink behaviour and its application to suibokuga-like 3d rendering of trees. **The Journal of Visualization and Computer Animation**, [S.l.], v.10, n.1, p.27–37, 1999.

Apêndice

```
// Evaluate this node returning the code generated
virtual string Eval(const NodeGraphEditor& nge, int indentLevel, int evalPort = 0, string scope = ""){
  // Begin with the parent eval (a comment indicating the node called)
  string code = Node::Eval(nge, indentLevel);

  // Define the actual level of indentation
  string ind = string(indentLevel*2, ' ');
  mScope = scope;
  //-------------------------------------------------------------
  // Get the information about nodes and so on
  int inIfPort;
  Node* inIf    = nge.getInputNodeForNodeAndSlot(this, 1, &inIfPort);
  ImVector<Node*> outputThenNodes = ImVector<Node*>();
  nge.getOutputNodesForNodeAndSlot(this, 0, outputThenNodes);
  ImVector<Node*> outputElseNodes = ImVector<Node*>();
  nge.getOutputNodesForNodeAndSlot(this, 1, outputElseNodes);
  //-------------------------------------------------------------
  // Check if there is a node connected to it
  if (inIf) {
    string varCondition = "out_" +inIf->getNameOutSlot(inIfPort)+ "_" +
                          std::to_string(inIf->mNodeId) + "_" + std::to_string(inIfPort);
    if(MustValidate(inIf->mScope, scope))
      code += inIf->Eval(nge, indentLevel, inIfPort, scope); // Here the variable out_portName_nodeID_port must be set

    code += ind+ "if(" +varCondition+ "){\n";
    if (outputThenNodes.size() > 0) // If there is a link for THEN
      for(Node* outThen:outputThenNodes)
        code += ind+ outThen->Eval(nge, indentLevel+1, 0, this->mScope + "S"+std::to_string(outThen->mNodeId)+"S");

    code += ind+ "} else {\n";
    if (outputElseNodes.size() > 0) // If there is a link for ELSE
      for(Node* outElse:outputElseNodes)
        code += ind+ outElse->Eval(nge, indentLevel+1, 0, this->mScope + "S"+std::to_string(outElse->mNodeId)+"S");

    code += ind+ "}\n";
  }
  return code;
}
```

Figure 1: Código da função *Eval* do nó *Conditional*, usada na geração do código em *C++* para exportação ou compilação.

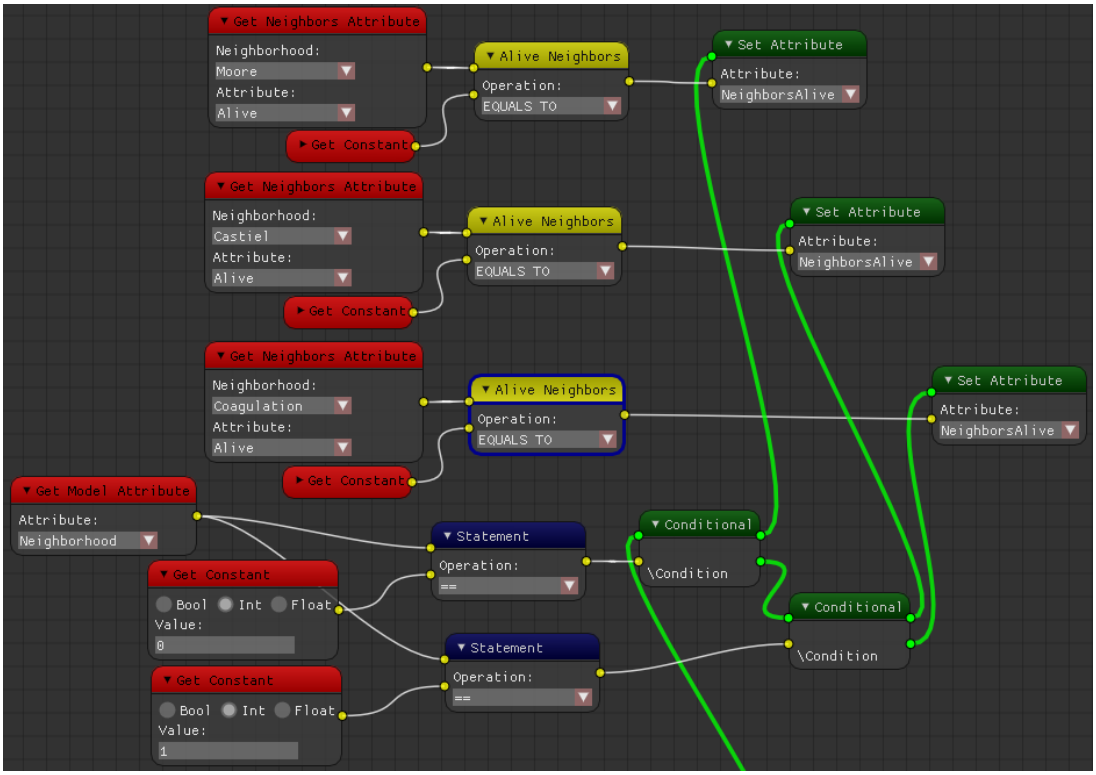


Figure 2: Esquema utilizado para realizar a troca de vizinhanças dinamicamente. É realizada uma demultiplexação do atributo de modelo *Neighborhood*, e de acordo com o valor (0, 1 ou 2), uma determinada vizinhança é considerada para determinar o valor do atributo *NeighborsAlive*. No resto do grafo, este atributo deve ser utilizado em vez do cálculo direto da vizinhança.

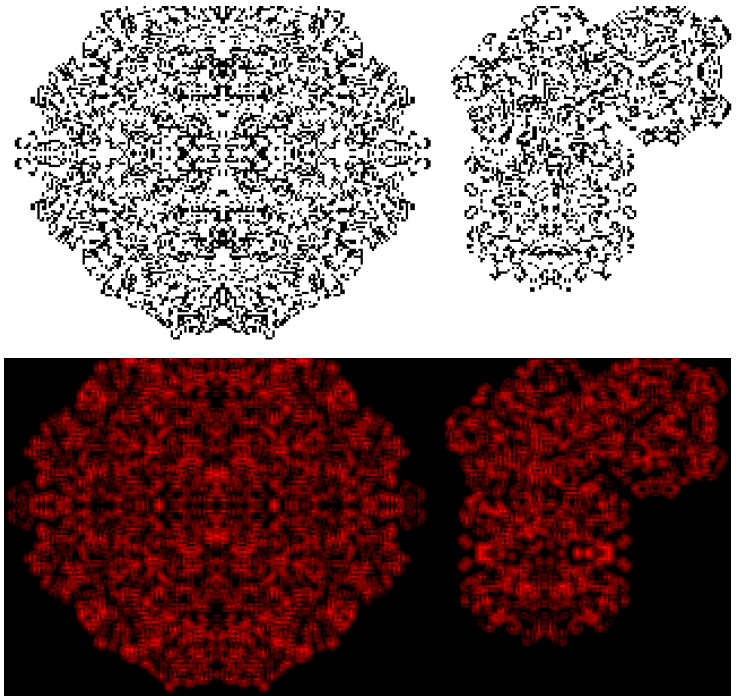


Figure 3: Comparação entre as visualizações *AliveView* e *DecoratedView* no *GoL*.

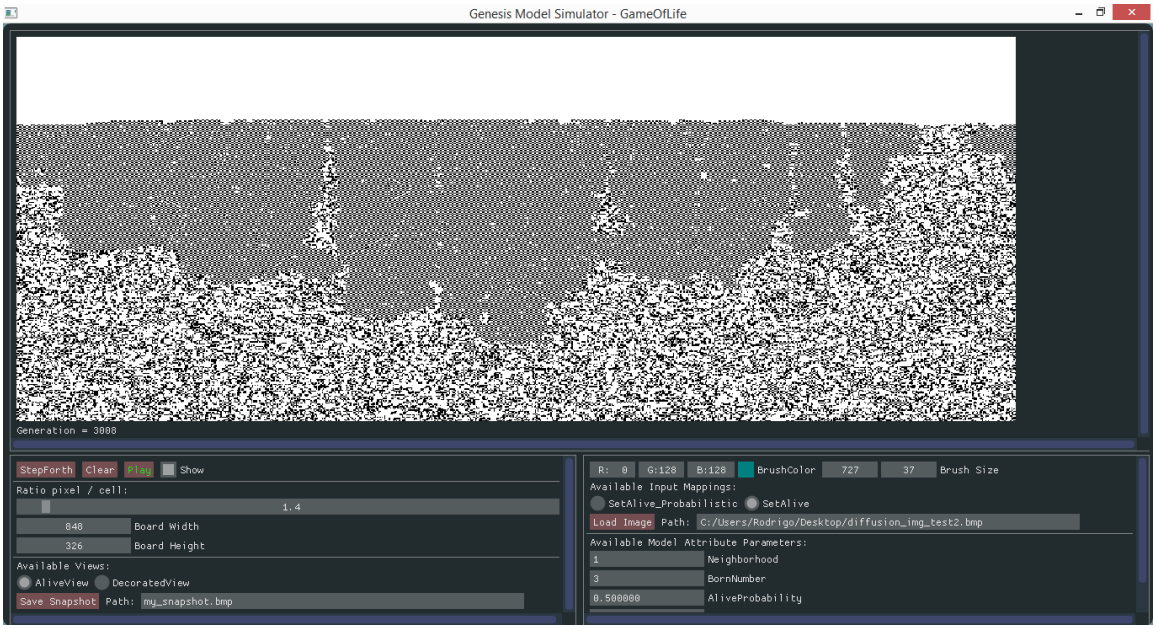


Figure 4: Variação do *GoL*, usando a vizinhança *Castiel*. Trocando o *UnderPopulation* por 1, é possível observar uma evolução do caos a ordem, e de volta ao caos, caso seja perturbado.