CEFET/RJ - CENTRO FEDERAL DE EDUCACAÇÃO TECNOLÓGICA CELSO SUCKOW DA FONSECA

Uma DSL para facilitar o uso de Cellular Automata em GPU

Eduardo Freitas Moura

Prof. Orientador:

Renato Campos Mauro

Rio de Janeiro,

13 de Junho de 2023

CEFET/RJ - CENTRO FEDERAL DE EDUCACAÇÃO TECNOLÓGICA CELSO SUCKOW DA FONSECA

Uma DSL para facilitar o uso de Cellular Automata em GPU

Eduardo Freitas Moura

Projeto final apresentado em cumprimento às normas do Departamento de Educação Superior do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

Prof. Orientador:
Renato Campos Mauro

Rio de Janeiro,
13 de Junho de 2023

DEDICATÓRIA

Para minha mãe e para meu pai.

RESUMO

Cellular Automata são excelentes para simulações e são estudadas a partir de muitos ângulos diferentes. Contudo, a simulação, a qual ocorre normalmente através da CPU do computador, traz problemas e dificuldades desnecessárias. Para facilitar pesquisadores e programadores que não sabem programar diretamente para a GPU, esse artigo propõem a criação de uma DSL, uma linguagem de domínio específico. A ideia é inserir o usuário em um domínio comum e, desse modo, facilitar sua especificação para uma simulação e executá-la em shader, o qual permite executar códigos personalizados diretamente na GPU. São exemplificados simulações de automata como o Jogo da Vida e modelos de Predador e Presa, com intuito de provar a eficiência da ferramenta DSL. Além disso, esse trabalho apresenta outros trabalhos que compartilham ou possuem propostas parecidas.

Palavras-chaves: Cellular Automata; Autômato Celular; Game of Life; Predador-Presa; DSL; shader; GLSL

Sumário

1 Introdução **1**

2 Fundamentação Teórica **3**

2.1 Autômatos Celulares 3

2.2 O Jogo da Vida 4

2.2.1 Representação em código 4

2.2.2 Demonstração Do Jogo da Vida 5

2.3 Modelos de predador e presa 6

2.3.1 Representação em código 7

2.3.2 Demonstração Do Predador-Presa 9

3 Materiais e Métodos **14**

3.1 DSL: Domain Specific Language 14

3.2 GLSL e Shaders 15

4 Trabalhos relacionados **17**

4.1 CAOS: uma linguagem de domínio específico para autômatos celulares 17

4.1.1 Declarations 18

4.1.2 Cells 19

4.1.3 Agents 19

4.1.4 Observers 20

4.2 CAOS e o Jogo da Vida 20

4.3 Simular um AC por shader 22

5 Desenvolvimento **25**

5.1 Funcionamento da DSL 25

5.2 Classes da DSL 27

6 Conclusão e Trabalhos Futuros **33**

Lista de Figuras

FIGURA 1:	Estado inicial do GameOfLife	6
FIGURA 2:	Estado Final do GameOfLife	6
FIGURA 3:	Estado inicial do tabuleiro do modelo Predador-Presa	10
FIGURA 4:	Predadores se alimentando de presas e reproduzindo	11
FIGURA 5:	Predadores morrem de fome e presas sobreviventes começam a se multiplicar	12
FIGURA 6:	Presas voltam a se reproduzir em massa e poucos predadores	13
FIGURA 7:	Gramática CAOS simplificada	18
FIGURA 8:	Sintaxe da seção de declarações	18
FIGURA 9:	Sintaxe da seção de comportamento	19
FIGURA 10:	Diagrama de classes da DSL	28

Capítulo 1

Introdução

Cellular Automata, ou autômatos celulares, são modelos simples e formais de computação que exibem comportamentos complexos de organismos auto-reproduzíveis. Originados por John von Neumann, um autômato celular (AC) é um sistema matemático discreto, que consiste em uma grade regular de células [Chen, 2009]. Cada célula possui determinadas propriedades e estados. Os estados são atualizados em passos discretos de tempo de acordo com regras de evolução definidas, as quais envolvem o estado atual da célula ou o estado de suas vizinhas. Posteriormente, físicos, biólogos e cientistas começaram a estudar autômatos celulares com o objetivo de modelar sistemas em seus respectivos domínios específicos [Sarkar, 2000]. Nos dias de hoje, ACs estão sendo estudados a partir de muitos ângulos diferentes. A relação dessas estruturas com problemas existentes está sendo constantemente buscada e descoberta.

Por serem ótimos meios de modelar sistemas, cientistas e pesquisadores também buscam exibir autômatos graficamente com intuito de analisar seu comportamento de forma visual. Contudo, a simulação, a qual ocorre normalmente através da CPU (Unidade Central de Processamento) do computador, traz alguns problemas. A exigência sobre o processador da máquina será enorme e capaz de demorar bastante tempo para a renderização de um modelo mais complexo de AC. Essa queda de performance acontece pois um processador é limitado a executar e renderizar apenas um pixel por vez [Patricio and Lowe, 2015]. Durante o processamento de imagens serão aplicadas várias operações a um conjunto de imagens, os resultados são guardados em um objeto e a imagem é exibida. Isso requer uma quantidade significativa de computação e movimentação de dados antes que a imagem final seja copiada para o objeto de imagem e para a memória de textura do dispositivo gráfico. Por esse motivo, a simulação de eventos com técnicas gráficas convencionais, ou seja, através da CPU, não se mostram muito eficientes para estes sistemas.

A solução desse problema de performance vem com o uso de shaders, ou GLSL (OpenGL Shading Language). Se trata de uma linguagem de alto nível, a qual oferece aos desenvolvedores um controle mais direto do pipeline de gráficos sem ter de usar linguagens específicas de hardware [Wolff, 2011]. Como a GPU é capaz de executar fragmentos(pixels) simultaneamente,

o processamento acontece em tempo real, garantindo flexibilidade para criar novos efeitos visuais. Além disso, o processamento sobre a CPU diminui consideravelmente, o que deixa mais recursos disponíveis para outras operações.

Apesar dessas vantagens, é senso comum entre programadores gráficos que o uso da GLSL é bem difícil. Os conceitos de um shader são complexos e envolvem conhecimentos sobre a placa gráfica, álgebra linear, geometria, mapeamento de textura e iluminação. A sintaxe tem regras rigorosas e não é similar a linguagens convencionais, podendo ser confusa a novos usuários. São tantas nuances sobre o uso dessa linguagem que fazem com que muitos pesquisadores, mesmo conhecendo seus pontos positivos, a evitem.

Este artigo, olhando para esse cenário, busca diminuir os problemas que programadores novatos tem com o uso da linguagem GLSL. Para esse fim, é proposto uma ferramenta capaz de abstrair um modelo de autômato celular em um código shader, que virá a ser executado em um placa gráfica. Para isso, será utilizada uma DSL, uma linguagem de domínio específico, que se trata de uma linguagem de programação de expressividade limitada focada em um domínio particular [Fowler and Parsons, 2011]. A ideia é inserir o usuário em um domínio comum, para esse ter facilidade de compreender a sintaxe fornecida e irá aprender rapidamente [Karsai et al., 2009] a utilizá-la. Em termos técnicos, a partir do conhecimento da composição de um autômato celular (tabuleiro, estados, vizinhança, comportamento, entre outros), a DSL irá poder relacionar a abstração do AC a um código equivalente para execução em placa em gráfica. Especialistas irão especificar a lógica de suas simulações e a linguagem de domínio específico, por sua vez, fornecerá respostas e abstrações que possibilitariam a criação de um código com base no modelo da simulação. Desse modo, mesmo não tendo conhecimentos sobre shaders ou GPU, um usuário pode usufruir dessa tecnologia.

Esse artigo é organizado em mais seis seções, além dessa introdução. A seção 2 compreende os conhecimentos teóricos, os quais são, fundamentais para esse trabalho, sobre AC e suas simulações. Na seção 3, há a introdução dos métodos e materiais que irão vir a ser utilizados neste trabalho. Em Trabalhos Relacionados, correspondente a seção 4, são introduzidos outros projetos e trabalhos, que possuem similaridade com a proposta desse artigo, com intuito de usá-los como inspiração. A seção 5, descreve como ocorreu a criação e desenvolvimento da DSL. Por fim, na seção 6 será realizada a conclusão do trabalho e conceitos para futuros trabalhos.

Capítulo 2

Fundamentação Teórica

Nessa seção de fundamentação teórica serão expostos os conhecimentos necessários para formar a base teórica desse artigo. Primeiro, o foco será de autômatos celulares e os motivos que o fazem ser ótimos para modelar simulações. Em seguida, é exemplificado alguns modelos famosos de autômatos celulares como o Jogo da Vida de John Conway e sistemas de predador-presa de Lotka e Volterra.

2.1 Autômatos Celulares

Uma das abordagens ao se tratar de modelagem e simulação de sistemas é o uso de autômatos celulares ou "AC"para facilitar. Criado por John von Neumann como modelos formais de organismos auto reprodutores, um AC tem como seu principal propósito trazer o rigor do tratamento axiomático e dedutivo para sistemas naturais complicados [Sarkar, 2000]. Em termos técnicos, um *Cellular Automaton* é um modelo de sistemas formados por objetos "célula"[Shiffman, 2012], os quais tem as determinadas características:

- Cada célula está em uma posição de um tabuleiro, ou *grid*.
- Cada célula recebe um número finito de estados (simples ou complexos) os quais se alteram, ou "evoluem", a partir de uma passagem temporal.
- Cada célula tem uma vizinhança, a qual normalmente corresponde a uma lista das células adjacentes.

Então, em um AC, há uma rede de células homogêneas as quais, com o decorrer do tempo, vão interagir entre si e alterar seus estados de acordo com sua vizinhança e regras de comportamento. Esses modelos de simulação podem ser simples ou complexos dependendo da quantidade de células e da complexidade dos estados definidos. Além disso, é também caracterizada por uma execução extremamente rápida e paralelismo inerente, o que permite o uso de modelos grandes e paralização simples [Ullrich and Lückerath, 2020]. Portanto, essas vantagens possibilitam o uso de autômatos celulares em ambientes simulados de alta performance.

2.2 O Jogo da Vida

Criado por John Horton Conway em 1970, o jogo da vida é um autômato celular capaz de representar o ciclo de vida de uma célula em um tabuleiro. Teoricamente, o jogo feito por Conway possui o poder de uma Máquina de Turing universal, ou seja, qualquer coisa que pode ser computada algoritmicamente pode ser computada dentro do jogo da vida [Ullrich and Lückerath, 2020]. A implementação desse autômato servirá como base na criação do *framework* proposto por este artigo.

O *Game of Life* é um jogo sem jogadores no qual a sua evolução ocorre com base no seu estado inicial. Cada quadrado (ou célula) do *grid* possui dois estados possíveis, vivo ou morto. Além disso, cada célula interage com seus vizinhos, ou seja, células adjacentes verticalmente, horizontalmente e diagonalmente. A cada passo temporal do programa, Conway definiu que serão aplicadas três "regras genéticas"para nascimento, morte e sobrevivência.

As leis, ou regras, de Conway são relativamente simples e são ativadas com base nos vizinhos de cada célula. A primeira regra diz que qualquer célula viva com mais de três vizinhos vivos morre por superpopulação. A segunda, afirma que qualquer célula viva com menos de dois vizinhos vivos morre por subpopulação. Por fim, a terceira lei conclui que qualquer célula morta com três vizinhos vivos se torna viva por reprodução [Gardner, 1970].

2.2.1 Representação em código

Agora que a teoria por trás do Jogo da Vida foi detalhada, se torna preciso analisar o código em si e como é a arquitetura de um autômato celular. Pela implementação do *Game of Life* podemos obter como são definidas as células, seus respectivos estados e as regras de Conway. Foi utilizado como base da implementação o código em Processing de Miles Berry [Miles Berry, 2020].

```
int size = 5; // Tamanho celula
int col = 600/size; // Tamanho coluna
int row = 600/size; // Tamanho linha
int [][] grid = new int[col][row];
```

Esse trecho de código define a montagem do tabuleiro. Baseado no tamanho da tela e no tamanho da célula é definido o tamanho das colunas e linhas, criando assim o *grid* para o Jogo

da Vida.

```
if (status == 1 && vizinhos > 3) return(0); //superpopulacao
else if (status == 1 && vizinhos < 2) return(0); //subpopulacao
else if ( status == 0 && vizinhos == 3) return(1); //reproducao
else return(status);
```

Podemos representar as regras do jogo através de simples condições e retornos booleanos. Todos os nascimentos e mortes, definidos por status 1 e 0 respectivamente, ocorrem simultaneamente constituindo um processo evolutivo conforme as gerações passam.

```
for (int y = 1; y < row - 1 ; y++) {
   for ( int x = 1; x < col - 1; x++) {
       int vizinhos = contaVizinhos(x,y);
       next_grid[x][y] = gameOfLife(grid[x][y],vizinhos);
    }
}
grid = next_grid;
drawGrid();
```

O código acima representa a leitura de célula por célula do tabuleiro. A cada iteração é contado cada vizinho adjacente a célula específica. Esse valor, junto com o próprio tabuleiro, são passados como parâmetros para a função *gameOfLife* a qual aplica as regras do jogo. O retorno será um novo tabuleiro, atualizado com as mudanças no estado de cada célula. Em seguida, o *grid* atualizado será desenhado novamente na tela.

2.2.2 Demonstração Do Jogo da Vida

Durante as "evoluções"do jogo, podemos observar as células morrendo e as novas gerações de células percorrendo o tabuleiro. Em um determinado momento, algumas células estarão agrupadas em certos padrões que não acionam nenhuma regra. Quando o programa atinge esse estado, não há novas gerações até executar novamente o *Game Of Life*. Essa aplicação pode ser compilada inúmeras vezes e sempre terá um padrão diferente do anterior. As figuras abaixo representam essas evoluções, mostrando o tabuleiro no começo da execução do programa e o mesmo tabuleiro poucos minutos depois.

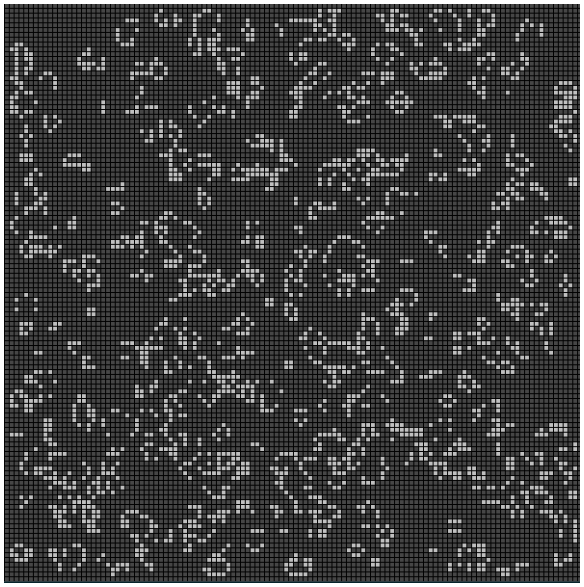

Figura 1: Estado inicial do GameOfLife

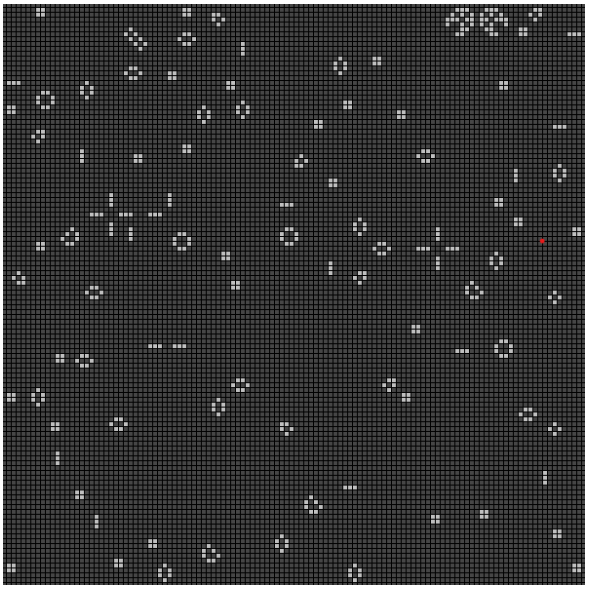

Figura 2: Estado Final do GameOfLife

Tornou-se comum desenvolver e simular comportamentos complexos de uma célula e seus relacionamentos com seus vizinhos. Isso define um contraste com os modelos de simulação, como o próprio Game of Life, o qual é constituído a partir de células que por sua vez têm comportamento e estrutura simples e ainda possuem dinâmicas intrincadas. Desse modo, é útil ter metodologias de modelagem e simulação que não são apenas fortemente fundamentadas em teorias matemáticas, mas também complementado com *frameworks* expressivos capazes de exibir com mais clareza e facilidade a evolução dessas células [Zhang and Sarjoughian, 2017].

2.3 Modelos de predador e presa

Os tipos de modelo predador e presa foram amplamente estudados no decorrer dos anos desde Lotka e Volterra, aqueles que primeiramente pensaram nesse modelo. O sistema predador-presa consiste em dois objetivos principais para seu estudo: o primeiro é a explicação das oscilações nas séries temporais das duas espécies, enquanto o segundo é a derivação dos estados estacionários de longo prazo alcançados pelo sistema [Farina and Dennunzio, 2008].

Em outros termos, o modelo retrata duas espécies que evoluem juntas e habitam o mesmo ambiente. Esse tipo de simulação irá simular a dinâmica de interação entre dois tipos de organismos com o passar do tempo, um fará o papel de presa e o outro de um predador [Obaid, 2013]. Alguns exemplos reais são: leão e zebras, ursos e peixes, gafanhotos e folhas. Um predador irá se alimentar de sua presa respectiva enquanto essa estará se reproduzindo com o decorrer do tempo. Desse modo, um predador morre se não se alimentar de uma presa e, caso

consiga capturar sua presa, como parte de sua evolução, se torna mais forte e se multiplica. De mesmo modo, a presa busca sempre se reproduzir e evitar a morte pelo predador [Obaid, 2013].

Embora equações diferenciais ordinárias sejam em muitos casos suficientes para atingir esses objetivos, elas têm a desvantagem de não fornecerem informações sobre a distribuição espacial das populações [Farina and Dennunzio, 2008]. O uso de autômatos celulares, os quais são ótimos para representação de modelos biológicos, se tornou ideal. Isso é possível pois, com o uso de AC, todos os estados da rede sejam atualizados de forma síncrona e homogênea por uma regra de evolução simples [Cattaneo et al., 2006]. Neste contexto, para alcançar o objetivo deste artigo, o modelo de predador e presa, da mesma forma que o Jogo da Vida, será utilizado para alcançar o objetivo proposto.

2.3.1 Representação em código

Utilizando o código de Alex Valente[Valente, 2018] como base, é analisado como a teoria de um modelo predador e presa é expressado em uma linguagem de programação. Assim como na implementação do Jogo da Vida, na seção 2.2.1, a implementação do modelo foi feita em processing.

Em primeira instância, deve-se especificar a presa e o predador, ou seja, especificar as funções de ação, movimento, atributos, entre outras declarações que cada tipo de célula vai precisar. Para facilitar, foi criada uma classe tanto para predador quanto presa. Além disso, para salvar a posição e os dados de todas as presas e predadores, foi declarado uma lista de array para cada classe de célula especificada.

```
ArrayList<Predator> preds = new ArrayList<Predator>();
ArrayList<Prey> prey = new ArrayList<Prey>();
```

Começando pelo predador, a classe *Predator* vai armazenar os dados e possuir as funções referentes ao predador. As mais importantes seguem abaixo. A função *Predator()* é o construtor responsável por inicializar uma célula predador. Ela irá aleatoriamente escolher uma posição no tabuleiro para designar o estado à célula e também define qual será a vida dessa célula em um intervalo estático. A função *vida()* funciona tal como uma ampulheta, a cada passagem de tempo ela irá retirar um ponto de *vida* do predador. O intuito é simular a fome, caso a *vida* de uma célula predatória chegue a zero, a mesma morrerá. Em contrapartida, a função *come()* é fundamental para o modelo, pois é ativada no instante que um predador se encontra com uma

célula de tipo presa. Primeiramente, toda a energia acumulada pela presa é transferida como *vida* para o predador. Em seguida, a presa "consumida"é removida do tabuleiro e um novo predador nasce em seu lugar.

```
class Predator
  Predator(){
    x = (int)random(width);
    y = (int)random(height);
    vida = (int)random(500, 1000);
  }
void vida()
    vida--;
    if(vida <= 0)
      Morreu = 1;

void come(int boost, int x, int y, int idx)
    vida += boost;
    prey.remove(idx);
    preds.add(new Predator(x, y));
```

A classe *Prey* vai armazenar os dados e possuir as funções referentes a presa. Da mesma forma que na classe *Preadator*, a função *Presa()* é o construtor responsável por inicializar uma célula tipo presa e também algumas variáveis para execução das regras. Uma célula presa não irá precisar consumir outra célula para se reproduzir, por isso sua reprodução deverá ser limitada utilizando as variáveis *vidaAux* e *ReprCount*. A função *vida()*, ao invés de diminuir a vida da célula com o tempo, irá aumentar a *vidaAux* até ser maior que a vida inicializada, quando isso ocorrer significa que uma presa possui energia suficiente para reprodução. Os contadores são atualizados para o processo ocorrer novamente. Além disso, para evitar uma multiplicação absurda de presas, uma presa irá morrer independente de ação predatória após sua terceira reprodução. Por fim, a função *temBebe()* vai ser chamada caso a variável *taGravida* não está zerada, e irá inserir uma nova presa no tabuleiro ao lado da sua célula presa mãe.

```
class Prey
  Prey(){
    x = (int)random(width);
```

```
    y = (int)random(height);
    vida = (int)random(350, 500);
    vidaAux = 0;
    ReprCont = 0;
    Morreu = 0;
  }
void vida()
    vidaAux++;
    if(vidaAux > vida){
      vidaAux = 0;
      taGravida = 1;
      ReprCont++;
      if(ReprCont > 2)
        Morreu = 1;
    }
void temBebe()
    taGravida = 0;
    prey.add(new Prey(x++, y--));
```

Esse modelo de predador e presa não terá muita diferença do Jogo da Vida a partir desse ponto. Temos estados com diferentes regras de evolução, mas a leitura do tabuleiro, dos vizinhos e a progressão de tempo é a mesma da do *Game of Life*. Será analisado a vizinhança de uma célula, as oito adjacentes, e a partir desse conhecimento ocorrerá a ativação das funções presentes nas classes de cada célula. Essas semelhanças entre a especificação de cada AC em código são o passo inicial para entender o mínimo necessário para definir um autômato.

2.3.2 Demonstração Do Predador-Presa

Ao iniciar a execução, temos uma quantidade igual de pixels verdes referentes as presas e de pixels vermelhos referentes ao predador. Cada pixel é uma célula no grid do autômato e irão se mover em direções aleatórias até que, no caso das presas, se reproduzam ou morram ou, no caso de um predador, coma a presa ou morra de fome.

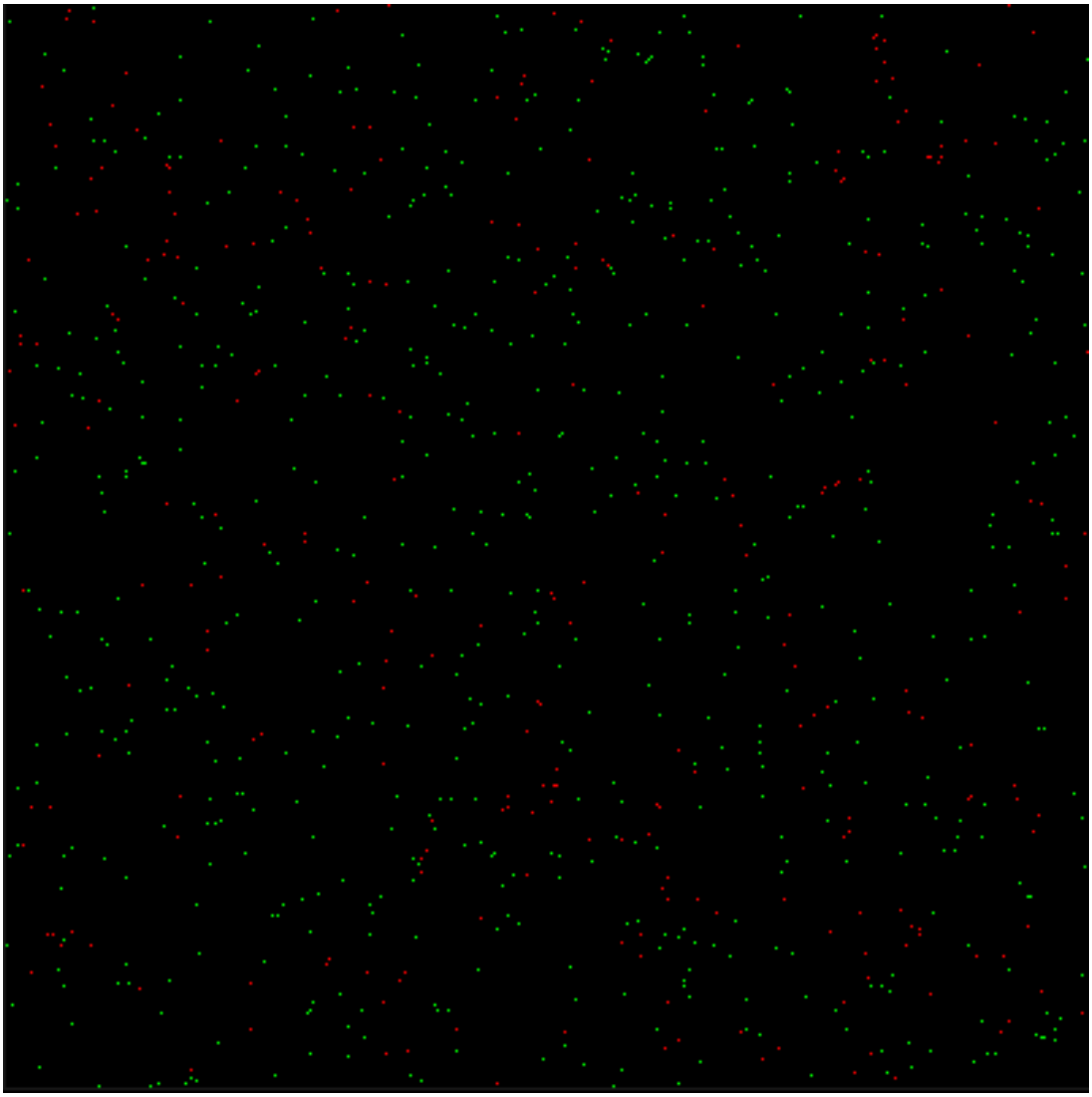

Figura 3: Estado inicial do tabuleiro do modelo Predador-Presa

Como as posições iniciais são aleatórias, é relativamente fácil notar que os predadores se espalham com mais facilidade. As células vermelhas irão se espalhar nesse primeiro momento e só irão sobrar algumas células verdes em pontos isolados do tabuleiro.

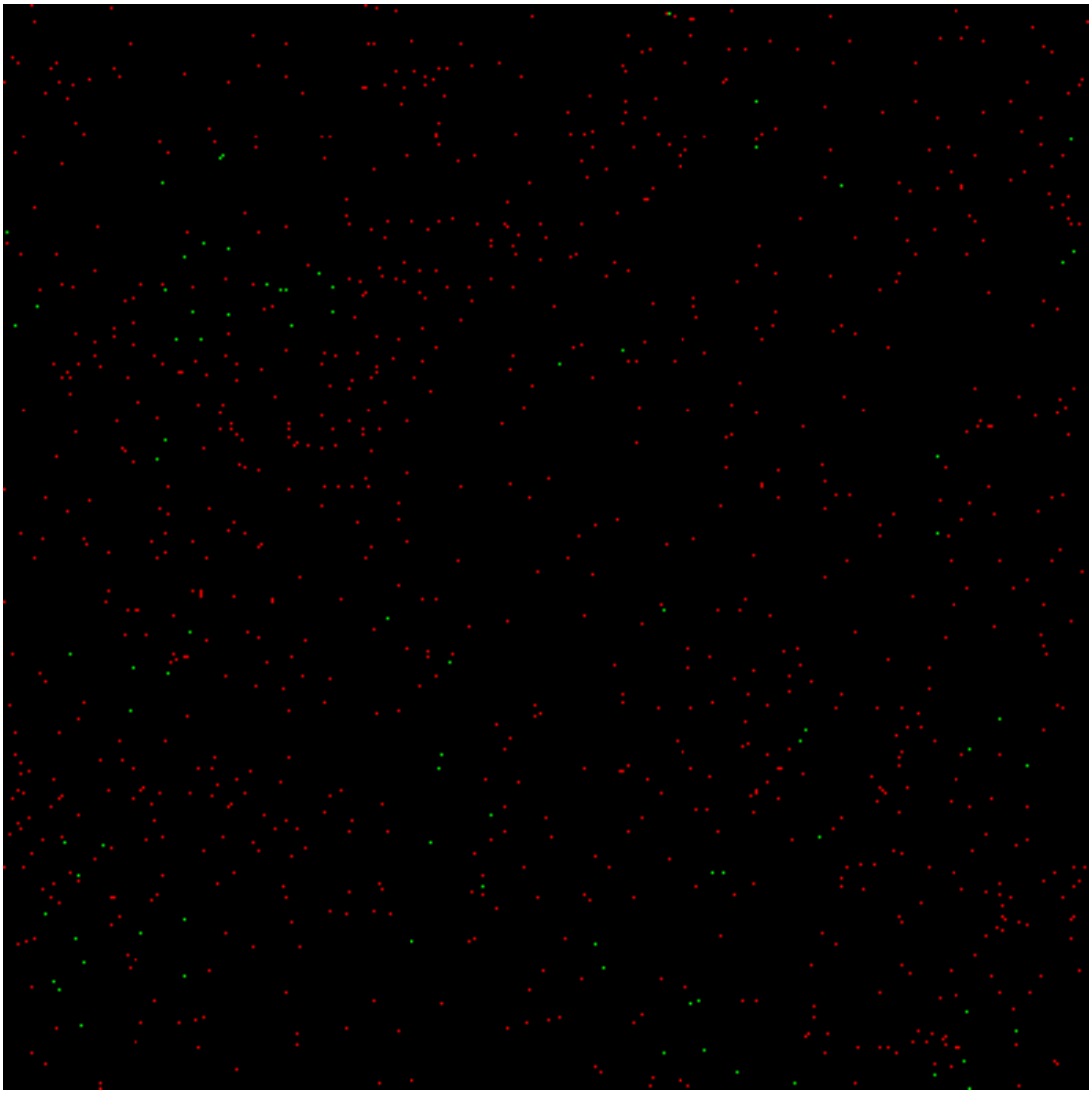

Figura 4: Predadores se alimentando de presas e reproduzindo

Eventualmente, os predadores, que estão em sua maioria, irão morrer de fome. Sem presas suficientes para eles, não terão como se alimentar e se reproduzir. Desse modo, a população de células vermelhas irá cair drasticamente, sobrando apenas aqueles mais próximos as presas remanescentes ou com maior valor de vida. Percebe-se que o tabuleiro está praticamente vazio, apenas com poucos pixels coloridos.

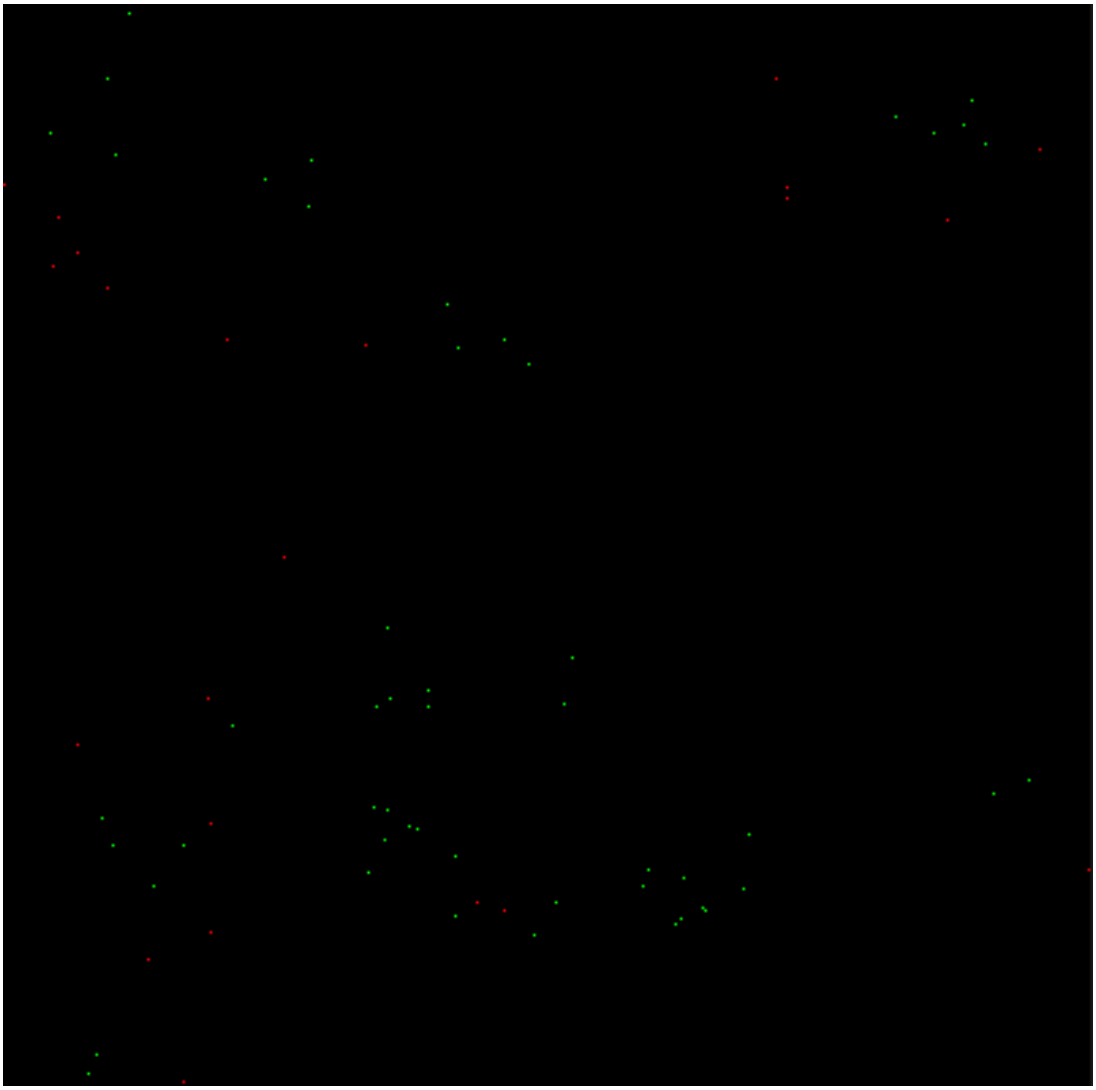

Figura 5: Predadores morrem de fome e presas sobreviventes começam a se multiplicar

Finalmente, com a população de predadores baixa, as presas terão espaço para se espalhar absurdamente pelo tabuleiro. Haverão pontos densos de pixels verdes, representando as colônias de células tipo presa que sobreviveram as primeiras etapas e agora estão se reproduzindo freneticamente. Contudo, o número de predadores que sobraram tem a chance de se alimentar e reproduzir novamente de forma fácil com o número elevado de presas. A imagem abaixo mostra a densidade de alguns pontos verdes e os pontos vermelhos se aproveitando pelas beiradas.

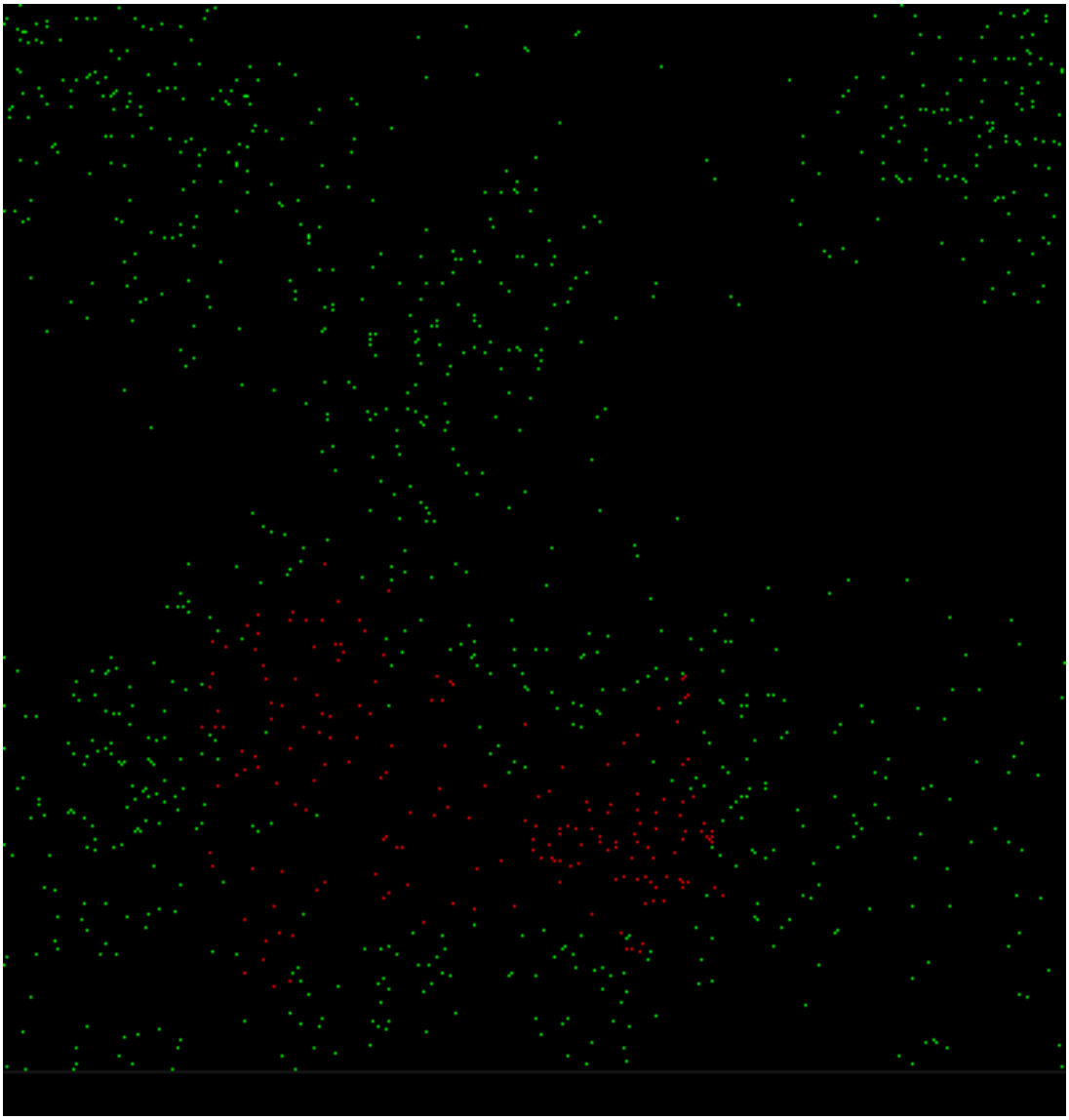

Figura 6: Presas voltam a se reproduzir em massa e poucos predadores

Essa simulação representa fielmente um modelo de predador e presa. Com comida, o número de predadores aumenta, até não haver mais suficiente para todos. Nesse momento, os predadores morrem e as presas conseguem aumentar seus números. Agora que há mais comida, os predadores tem como se reproduzir novamente e repetir o ciclo.

Capítulo 3

Materiais e Métodos

Na seção anterior, foi exposto uma base teórica para auxiliar na realização do objetivo final deste artigo. Agora, em Materiais e Métodos, será exposto algumas ferramentas e utilitários que são necessários para o projeto.

3.1 DSL: Domain Specific Language

Este trabalho busca projetar uma nova linguagem que permite modelar propriedades técnicas de forma mais simples e fácil, descrever e implementar soluções, e também até descrever o problema em questão. Para esse fim, deverá ser desenvolvida uma linguagem de domínio específico ou Domain Specific Language (DSL). Se trata de uma linguagem de programação de computador de expressividade limitada focada em um domínio particular [Fowler and Parsons, 2011]. De acordo com Fowler em seu livro Domain Specific Languages[Fowler and Parsons, 2011], uma DSL possui quatro elementos chaves para sua definição, esses são:

- Estrutura é projetada para facilitar o seu entendimento e ainda deve ser algo executável por um computador.
- É uma linguagem de programação e deve obter expressividade não só das expressões individuais definidas mas também dos conjunto de expressões formados.
- Deve suportar um mínimo de recursos necessários referentes ao seu domínio.
- Para ser uma linguagem útil, precisa ter foco em seu domínio.

Como uma DSL é utiliza uma linguagem específica a um domínio comum do usuário, esse terá enorme facilidade de compreender a sintaxe fornecida e irá aprendê-la mais rapidamente [Karsai et al., 2009]. Por exemplo, existe a necessidade de uma solução que capacite especialistas com o poder de especificar a lógica de suas simulações. As linguagens domínio específico irão fornecer respostas e abstrações que possibilitariam os especialistas de domínio a compreender a questão e criar facilmente um código para sua simulação. Porém, essas linguagens muitas

vezes se encaixam apenas a um problema de domínio bastante específico e podem ser usados por muitas pessoas nem flexíveis o suficiente para ser facilmente adaptado para domínios relacionados [Raja and Lakshmanan, 2010]. Dessa forma, para definir corretamente uma DSL, é primordial ter conhecimento do domínio escolhido.

Implementar uma nova linguagem a partir do zero permite um enorme grau de liberdade, agilidade e flexibilidade. No entanto, essas vantagens não são gratuitas: projetar uma sintaxe, um sistema de elementos, testar a consistência da linguagem, entre outras implementações.[Karsai et al., 2009]. Para certos casos, seria proveitoso reutilizar uma linguagem já existente ou buscar adaptar uma parecida, visto que criar uma nova DSL do zero é um processo que é custoso e consome muito tempo.

Para alcançar o objetivo proposto, essa linguagem, no contexto desse projeto, deverá ser capaz de abstrair um autômato celular e descrevê-lo em uma linguagem voltada para execução em GPU. Em termos técnicos, a partir do conhecimento da composição de um autômato celular (tabuleiro, estados, vizinhança, comportamento, entre outros), a DSL irá poder relacionar a abstração do AC a um código equivalente para execução em placa em gráfica.

3.2 GLSL e Shaders

Baseada na linguagem de programação C, o OpenGL Shading Language (ou GLSL) é uma linguagem de shading de alto nível, a qual é uma parte fundamental e integral do OpenGL [Wolff, 2011]. Essa linguagem de programação oferece aos desenvolvedores um controle mais direto do pipeline de gráficos sem ter de usar linguagens específicas de hardware. O processamento acontece em tempo real, garantindo flexibilidade para criar novos efeitos visuais e a CPU é aliviada de um processamento mais pesado.

O compilador do GLSL é construído dentro da biblioteca do próprio OpenGL, ou seja, em qualquer ambiente que possui o contexto do OpenGL, pode ser executado um shader [Wolff, 2011]. Esse processo envolve a criação de um objeto shader. Deve-se fornecer como *input* um código-fonte (como uma *string* ou conjunto de *strings*) para o objeto criado e, em seguida, solicitar ao objeto para compilar o código. Dessa forma, temos um código shader em execução.

Um shader, para computação gráfica, é também um conjunto de instruções, porém esses comandos são executados para todos os pixels da tela simultaneamente [Patricio and Lowe, 2015]. Com esse programa de computador, executado em GPU, pode-se criar efeitos especiais, produzir vários níveis de cor e fazer um pós-processamento de imagem. Em termos técnicos, o

programa shader trabalha como se fosse uma função, ela recebe uma posição, um *vertex*, e retorna uma cor, um *fragment*. Os Vertex shaders e pixel shaders dos chips gráficos 3D atuais são os responsáveis por uma ampla capacidade de programação para criar efeitos impressionantes ou simplesmente descarregar tarefas demoradas da CPU [Loviscach, 2004]. Com o uso de shaders, pode-se executar códigos personalizados diretamente na GPU, o que permite aproveitar o alto grau de paralelismo disponível nas GPUs modernas[Wolff, 2011].

Apesar dessas vantagens, é senso comum de programadores gráficos que o uso da GLSL é bem difícil, inclusive para aqueles experientes. As noções de shader são muito complexas, envolvem conhecimentos sobre a placa gráfica, álgebra linear, geometria, mapeamento de textura e iluminação. A sintaxe, a qual difere muito das linguagens de programação convencionais, tem regras rigorosas para a definição de instruções em código, logo, pode se tornar confusa ao usuário não familiar a ela. O hardware gráfico possui limitações e peculiaridades que podem prejudicar a execução de um shader, essas nuances precisam de um conhecimento detalhada da arquitetura do hardware. Além disso, não é possível depurar um shader passo a passo como em linguagens de programação tradicionais, o que torna a depuração um desafio também. Com prática, estudo e experiência, é possível superar esses desafios e adquirir proficiência em GLSL, mas nem sempre há tempo para isso.

Capítulo 4

Trabalhos relacionados

Nesta seção de trabalhos relacionados, como o próprio nome já diz, são exibidos outros projetos relevantes ao tema deste projeto. O intuito é usá-los como base, analisando brevemente suas arquiteturas e resultados. Para o contexto deste artigo, será apresentado a CAOS, uma DSL que auxilia na criação de autômatos celulares. Essa linguagem será a inspiração para a DSL proposta nesse trabalho, visto possuem propósitos similares.

4.1 CAOS: uma linguagem de domínio específico para autômatos celulares

Como mencionado anteriormente, autômatos celulares são ideais para modelagem de sistemas e fenômenos observados nas ciências naturais. Esse trabalho de modelagem de autômatos é tipicamente feito por cientistas, os quais são especialistas em suas respectivas áreas, contudo não necessariamente são programadores experientes. A programação de simulações complexas de forma eficiente transforma-se rapidamente em um empreendimento doloroso distraindo os aspectos interessantes da simulação problema em si.

A CAOS (Células, Agentes e Observadores para Simulações) [Grelck et al., 2007], com um intuito similar a este projeto, foi desenvolvida para auxiliar nas pesquisas e projetos destes cientistas que são pouco versados em programação e arquitetura de computadores. Se trata de uma linguagem de domínio específico a qual é feita sob medida para software de simulação de programação baseado no modelo de autômatos celulares. Ela fornece a cientistas com pouca experiência em programação suporte para rápida prototipagem de sistemas complexos.

A linguagem da CAOS foi baseada em uma gramática, a qual é organizada em três seções principais, que por acaso compõem o nome da linguagem: células, agentes e observadores [Penczek, 2006]. Nestas seções, estritamente pré-determinadas, será definido o *grid* do autômato, as células que irão o compor, o comportamento de cada célula, entre outras definições. Abaixo vemos um exemplo da gramática simplificada.

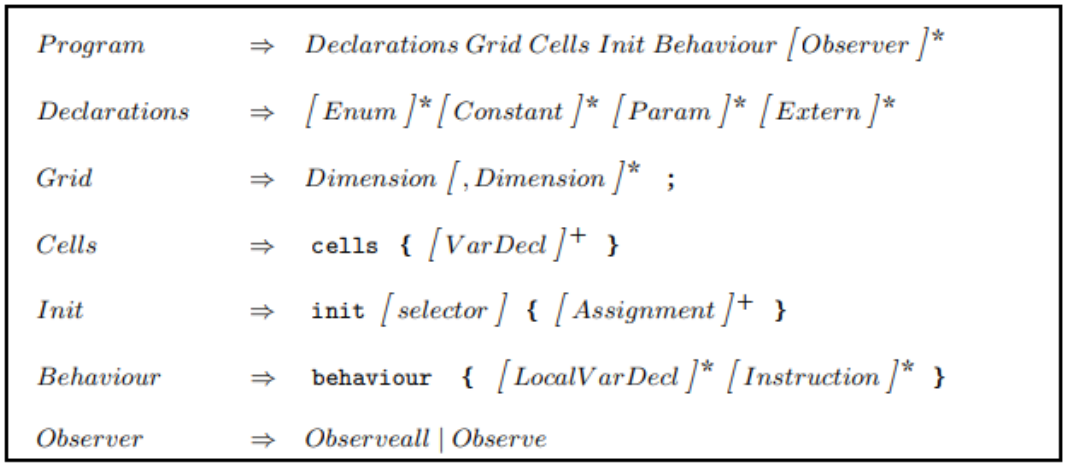

Figura 7: Gramática CAOS simplificada

Para uma explicação sucinta dos símbolos referentes ao símbolo inicial *Program*, as subseções a seguir dão profundidade sobre o símbolo *Declarations*, *Cells*, *Agents* e *Observers*.

4.1.1 Declarations

Declarations é o símbolo que vai ser responsável por declarar os objetos, constantes, parâmetros globais que irão ser utilizados no decorrer do programa. A CAOS não é uma linguagem muito eficiente em construir tipos e estruturas, então essa parte é de fato restrita [Grelck et al., 2007]. Os tipos aceitos para declaração de dados são apenas *bool*, *int* e *double*. A imagem abaixo mostra a sintaxe dessa seção.

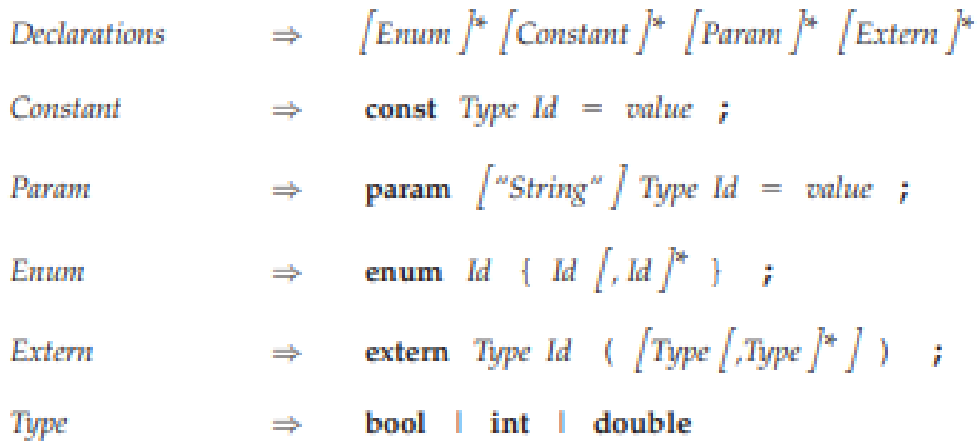

Figura 8: Sintaxe da seção de declarações

4.1.2 Cells

O símbolo *Cells* consiste basicamente em atribuir a uma célula suas declarações básicas. Por exemplo, a posição da célula no tabuleiro, a declaração de seus atributos, seu estado inicial e seu comportamento. Esta especificação de uma única célula complementa a definição do *grid* do AC, já que define a montagem dessas células uniformes no autômato celular. Pela gramática apresentada, pode-se notar que o símbolo *Cells* possui sua produção apontada para todos símbolos responsáveis por declarar esses valores.

O símbolo *Init*, o qual é responsável por definir o estado inicial do autômato, é utilizado para inicializar todos os componentes definidos como atributos da célula. Para evitar problemas na execução e situações não previstas, se faz necessário sempre inicializar as células.

Por fim, a especificação do *Behavior* representa uma parte fundamental de um AC, o comportamento que será seguido por cada célula. Nessa parte, os símbolos relacionados ao comportamento possuem familiaridade com blocos de códigos de controle de fluxo da linguagem C ou Java. Os atributos da célula também são utilizados para serem modificados de acordo com a lógica. A seguir, um exemplo dessa etapa da gramática.

$$
\begin{array}{lcl}
\mathit{Behaviour} & \Rightarrow & \textbf{behaviour}\ \{\ [\mathit{Type}\ \mathit{Id}\ [\ =\ \mathit{Expr}\ ]\ ;\ ]^*\ [\mathit{Instruction}\ ]^*\ \} \\
\mathit{Instruction} & \Rightarrow & \mathit{Assignment}\ |\ \mathit{Cond}\ |\ \mathit{ForEach}\ |\ \mathit{Switch} \\
\mathit{Assignment} & \Rightarrow & \mathit{Id}\ =\ \mathit{Expr}\ ; \\
\mathit{Cond} & \Rightarrow & \textbf{if}\ (\ \mathit{Expr}\ )\ \mathit{Block}\ \textbf{else}\ \mathit{Block} \\
\mathit{ForEach} & \Rightarrow & \textbf{foreach}\ (\ \mathit{Type}\ \mathit{Id}\ \textbf{in}\ \mathit{Set}\ )\ \mathit{Block} \\
\mathit{Switch} & \Rightarrow & \textbf{switch}\ (\ \mathit{Id}\ )\ \{\ [\mathit{Case}\ ]^+\ [\mathit{Default}\ ] \\
\mathit{Case} & \Rightarrow & \textbf{case}\ \mathit{CaseVal}\ [\ ,\ \mathit{CaseVal}\ ]^*\ :\ \mathit{Block} \\
\mathit{Block} & \Rightarrow & \mathit{Instruction}\ |\ \{\ [\mathit{Instruction}\ ]^*\ \}
\end{array}
$$

Figura 9: Sintaxe da seção de comportamento

4.1.3 Agents

Agents são bastantes similares ao símbolo *Cells* visto que também consistem em um conjunto de atributos. A cada passo temporal da simulação do autômato, um agente é associado a exatamente uma célula. Por outro lado, uma célula pode ser associada a um grande número de agentes.

Os agentes também possuem um comportamento, o qual é baseado em seu estado existente e no estado da célula em que reside. Além de atualizar o estado de uma célula, um agente pode

também decidir transitar para uma célula vizinha. Para complementar, os agentes também têm um tempo de vida, ao invés de se mudar para outra célula, os agentes podem decidir morrer e os agentes podem criar novos agentes.

4.1.4 Observers

Observers servem literalmente o propósito de seu nome. Basicamente, permite observar os valores de atributos específicos de células e agentes ou dados cumulativos sobre eles como por exemplo médias, mínimos ou somas. Essas informações são observadas em certos intervalos de tempo regulares da simulação ou logo após completar toda a simulação.

4.2 CAOS e o Jogo da Vida

Com o intuito de testar essa gramática, o CAOS será utilizado para fazer um protótipo de um Jogo da Vida. Esse autômato será ótimo como exemplo pois não possui muitos estados e também suas regras são bem definidas. Desse modo, o exemplo de código do Jogo da Vida visto anteriormente será o referencial para essa implementação.

Primeiramente, seguindo a lógica da gramática do CAOS, são feitas as declarações iniciais do programa e algumas definições. Para ser fiel a implementação interior, é definido as constantes do número de linhas e colunas, como 80 e 80, e também uma enumeração que representará os estados da célula, 0 e 1 para morto ou vivo respectivamente. Logo, a seção de declarações ficará assim:

```
enum MortoVivo [ 0 , 1 ] ;
const int coluna = 80 ;
const int linha = 80 ;
```

Com isso, a próxima etapa será montar um *grid*, um tabuleiro, no qual corresponde a área de ação do autômato. É almejado um quadrado oitenta por oitenta, como determinado pelas contantes nas declarações. Para isso, definimos cada eixo do tabuleiro, suas direções e seus limites. Haverá algo parecido como:

```
grid X:0..colunas:left<.>right:cyclic,
     Y:0..linha:up<.>down:cyclic;
```

Utilizando da própria definição do símbolo *Grid* da gramática, percebe-se que o eixo X é definido pelo índice 0 ao índice equivalente ao número de linhas, da esquerda à direita. Da mesma forma, o eixo Y, agora de cima para baixo, é definido entre os índices 0 e linha. Além disso, ambos eixos possuem fronteiras e limites, como é indicado pelo símbolo *Boundary*. Ele permite definir se o tabuleiro será cíclico (*cyclic*) ou estático (*static*), se uma célula no extremo direito vai conseguir interagir ou não com outra que está no extremo esquerdo do tabuleiro. Para o *Game Of Life* os limites são cíclicos, logo as bordas dos eixos X e Y recebem o parâmetro *cyclic* no final de sua definição.

O próximo passo é definir a estrutura da célula em si, seus atributos principais. O Jogo da Vida só possui dois tipos de estados e cada célula só possui um deles por vez, portanto a declaração do enumerador MortoVivo é essencial para essa parte. Uma estrutura simples é estabelecida com o intuito de armazenar qual o estado atual de uma célula. A variável status irá, como um dado do tipo MortoVivo, sempre será 0 ou 1.

```
celula{
    MortoVivo status
}
```

Por fim, o comportamento da célula é detalhado. Não será complicado, visto que o símbolo *Behavior* possui algo parecido com blocos de controle similares a linguagens comuns de programação. Além disso, as regras do jogo da vida foram definidas em código anteriormente (superpopulação, subpopulação e reprodução) e a contagem dos vizinhos também deve ser exibida. Basicamente, o comportamento do *Game Of Life* representado pela gramática do CAOS se dá por:

```
behavior {
    int soma_vizinhos;
    soma_vizinhos = status[up^left] + status[up] + status[up^right]
                  + status[left] + status[right]
                  + status[down^left] + status[down] + status[down^right]
    if(status = 1)
      if(soma_vizinhos > 3 ) status = 0 ;
    else
   if(soma_vizinhos < 2 ) status = 0;
```

```
    else
        if(soma_vizinhos = 3) status = 1;
}
```

Ao definir todas as direções, é possível buscar os status de todas as casas adjacentes a uma célula no tabuleiro. Somamos os valores de seus status e obtemos a quantidade de vizinhos vivos ao redor da célula principal. Com blocos de condições simples, é possível aplicar as regras definidas por Conway e atualizar o valor de uma célula de acordo.

Com isso, em poucos minutos, a linguagem do CAOS conseguiu criar uma abstração completa do Jogo da Vida. Foi comprovado que essa DSL é capaz de representar um tabuleiro, uma célula e o comportamento que rege um autômato. Contudo, o que a DSL proposta nesse trabalho busca, é justamento utilizar da mesma ideia, porém voltada à gerar um código voltado para execução em GPU.

4.3 Simular um AC por shader

Primeiramente, é necessário entender como um automato celular pode ser representado utilizando shaders. Esse conhecimento irá possibilitar a construção da nossa DSL, visto que o seu papel será traduzir um autômato para a linguagem GLSL. Por ser um autômato celular relativamente simples de implementar e bem difundido, o Jogo da Vida de Conway em shader será apresentado como exemplo.

Existem várias implementações do *Game of Life* com o uso de shaders pela internet. Pelo Shadertoy [ShadertoyBETA], foi encontrado um código de 2016 feito pelo usuário Chronos. Titulada como "Introdução ao Jogo da Vida", essa implementação de shader é ideal pois foca em ensinar pessoas que são razoavelmente novas em escrever shaders e pode servir como ponto de partida para implementações mais elaboradas [Chronos, 2016]. Nesse shader, haverá basicamente duas funções: a função principal (ou *main*) será responsável por mapear os pixels e exbi-los na tela; e uma outra função que será responsável por computar as regras do Jogo da Vida.

Dentro da função principal, como pretende-se ler a textura e mapear a janela inteira, precisamos dimensionar ambas as nossas coordenadas para mapear do intervalo 0 ao 1. Desse modo definiu-se o *vec2 uv* utilizando as coordenadas do fragmento e a resolução da tela no ponto *xy*.

```
vec2 uv = fragCoord.xy / iResolution.xy;
```

Em seguida, os dados da outra função são lidos através do *iChannel0*, uma funcionalidade do Shadertoy que é basicamente um canal que permite trazer os dados de um *buffer*, o qual representa a função de regras, e definir uma textura. O primeiro parâmetro da função textura é o *iChannel0*, para definir a textura que equivalerá ao estado inicial das células e os dados resultantes da função de regras. Por sua vez, o segundo parâmetro é a o vetor *uv* definido anteriormente.

```
vec4 color = texture(iChannel0, uv);
```

Por fim, a linha de código abaixo representa a cor da célula, em termos do Jogo da vida, branco ou preto, vivo ou morto, respectivamente.

```
fragColor = vec4(color.xyz, 1.0);
```

Na função onde é executado as regras do *Game of Life*, representada no *BufferA* do Shadertoy, há a maior parte do código. Será necessário, em primeira instância, novamente definir um vetor *uv* para definir as coordenadas da célula e haverá um variável temporária *color* para o valor do estado da célula.

```
fvec2 uv = fragCoord.xy / iResolution.xy;
vec3 color = vec3(0.0);
```

Como no Jogo da Vida, é necessário contar a vizinhança de cada célula. Isso é representado por um contador *neighbors*, iniciado com o valor nulo, e dois laços para leitura dos oito vizinhos adjacentes da célula em questão. O *vec2 offset* representa a posição do vizinho nessa volta do laço, que, ao ser somado com o vetor *uv* e usando a função *texture* mais uma vez, pode-se saber o estado da célula vizinha. Esse valor é somado ao contador de vizinhos.

```
float neighbors = 0.0;
for(float i = -1.0; i <= 1.0; i += 1.0)
{
    for( float j = -1.0; j <= 1.0; j += 1.0)
    {
        vec2 offset = vec2(i, j) / iResolution.xy;
   vec4 lookup = texture(iChannel0, uv + offset);
        neighbors += lookup.x;
```

```
    }
}
```

Em seguida a declaração do *float cell* é equivalente a amostragem do pixel atual, da célula atual. Essa variável é operada junto com a contagem de vizinhos para determinar se a célula está viva ou morta neste iteração.

```
float cell = texture(iChannel0, uv).x;
```

A próxima parte do códgio, representa as regras do Jogo da Vida. Como as mesmas foram apresentadas no tópico 2.2, não serão explicadas novamente. Contudo, deve-se atentar com o retorno de *color* como um *vec3*.

```
if(cell > 0.0) {
    if(neighbors >= 3.0 && neighbors <= 4.0) {
        color = vec3(1.0);
    }
} else if(neighbors > 2.0 && neighbors < 4.0) {
    color = vec3(1.0);
}
```

Para obter qualquer comportamento interessante, é preciso de um comportamento não uniforme para as condições de partida. Uma maneira simples de fazer isso é alimentar a textura de ruído no *buffer* para o primeiro quadro. Desse modo, o Jogo da Vida pode começar.

```
if(iTime < 1.0) {
    color = vec3(texture(iChannel1, fragCoord.xy / iResolution.xx).x);
}
fragColor = vec4(color, 1.0);
```

Capítulo 5

Desenvolvimento

Utilizando como base as implementações de autômatos celulares anteriormente definidas, tanto em processing quanto em shader, e utilizando-se da gramática definida pela linguagem CAOS, é possível extrair informações fundamentais para a linguagem proposta neste artigo. Analisando o modelo de predador e presa e o Jogo da Vida, percebe-se um padrão na definição de um AC. As declarações, as quais dizem respeito aos estados de cada célula, e as suas regras de evolução (seu comportamento) correspondem ao diferencial de um autômato para o outro.

Portanto, essas definições são julgadas principais e que precisam que seus valores sejam entrados pelo usuário. Algumas definições, como funções envolvendo vizinhos da célula e tamanho do grid, podem ser estáticas e também incorporadas de forma integral na DSL.

5.1 Funcionamento da DSL

Como a DSL irá funcionar na prática? Um usuário, primeiramente, terá de instanciar o objeto da classe DSL, que representa um programa o qual será gerado pelo shader. Em seguida, também será preciso declarar os operadores que irão ser usados em condições para as regras de evolução do AC. Em seguida, é preciso definir cada estado possível e qual virá a ser a função de vizinhança que virá a ser utilizada. Por fim, basta o usuário inserir as regras utilizando-se dos operadores, valor dos estados e retorno da função de vizinhança escolhida. Além disso, é essencial definir uma ação que virá a ser executada quando uma condição for verdadeira. Desse modo, um usuário terá todo o necessário para gerar seu código shader. Abaixo, um exemplo da chamada da classe:

```
const dsl = new DSL();
```

A DSL oferece suporte para operadores lógicos e relacionais. Entre os operadores lógicos, temos o AND e OR para condições lógicas. Para os relacionais, temos o símbolo para igual (EQ), maior que (GT), menor que (LT), maior igual (GE), menor igual (LE) e não igual (NE). A DSL, em seu estado atual, não possui operadores aritméticos e também não dispõem de todos

os operadores lógicos e relacionais para lógicas mais avançadas. Só será preciso declarar como constante os operadores que virão a ser utilizados nas regras, como consta no exemplo abaixo.

```
const { AND, EQ, GT, LT, LE, GE } = dsl.operators();
```

Durante a declaração dos estados de uma célula, a DSL permite que sejam armazenadas até quatro variáveis de estado. No shader gerado, essa variável será representada internamente com o tipo *vec4*. Portanto, para facilitar a geração do código, a linguagem irá obedecer essa limitação, permitindo no máximo quatro estados diferentes para as células do AC. O usuário irá conseguir declarar um estado com o uso do método *stateVar()*.

Também será necessário definir funções de agregação de vizinhança. A própria DSL terá em disposição algumas funções envolvendo os vizinhos de uma célula. Por exemplo, pode-se calcular quantos vizinhos existem ao redor de uma célula, a soma de todos seus valores, procurar um estado específico dentre células vizinhas, buscar o valor máximo ou mínimo, por aí vai.

Como exemplo, um usuário quer utilizar essa ferramenta para gerar um código em shader para o Jogo da Vida. Sabendo que ele já instanciou a classe DSL e declarou seus operadores, ele deve definir os estados da célula e a regra de vizinhança. Seguindo a lógica da implementação na seção 4.3, para o *Game of Life* será preciso apenas um estado, o estado "vivo"que se for maior que zero indica que a célula está viva e igual a zero indica que está morta. Para a vizinhança, é preciso retornar quantos vizinhos estão vivos ao redor. A função de vizinhança *sumNeighbors()* presente na DSL, é perfeita pois percorrerá as oito posições adjacentes a célula em questão e retornará o número de células vizinhas vivas. Lembrando que, como se trata de uma ferramenta simples, a vizinhança sempre será os oito quadrados ao redor de uma célula.

```
const vivo = dsl.stateVar();
const neighbors = dsl.sumNeighbors(vivo);
```

Por último, o usuário precisa definir uma lista de regras, as quais são basicamente compostas por duplas de elementos, uma condição e uma ação. As condições envolvem os operadores previamente definidos, as variáveis de estado, de vizinhança ou outra constante. Por outro lado, o elemento de ação está relacionado geralmente com a mudança de uma variável estado para a célula. Essa ação será executada caso a condição relativa a ela seja satisfeita e deve também respeitar as variáveis estado anteriormente declaradas.

Utilizando o mesmo cenário do Jogo da Vida e com os conhecimentos sobre o comportamento dessa simulação, pode-se obter um par de regras. Essas são:

```
dsl.rules([
    [ AND( GT(vivo,0.0) , AND(LE(neighbors,4.0), GE(neighbors,3.0))),
       vivo.setState(1.0)],
    [ AND( EQ(vivo,0.0) , AND(LT(neighbors,4.0), GT(neighbors,2.0))),
       vivo.setState(1.0)]
]);
```

A partir disso, a DSL conseguirá gerar um código para ser utilizado em shader representando fielmente o Jogo da Vida. Com essas inserções do usuário, o código gerado ficaria da seguinte forma:

```
/** Declarations **/
vec4 nextState = vec4(0.0);
vec4 neighborStatus= vec4(0.0);

/** Neighbor Status **/
neighborStatus[0]=sumNeighbors(0);

/** Rules **/
if(((currentState[0]>0.0)&&((neighborStatus[0]<=4.0)&&
(neighborStatus[0]>=3.0))))
    {nextState[0]=1.0;}
if(((currentState[0]==0.0)&&((neighborStatus[0]<4.0)&&
(neighborStatus[0]>2.0))))
    {nextState[0]=1.0;}
```

5.2 Classes da DSL

Nessa seção do desenvolvimento será dado uma visão mais técnica para a DSL. Foi exemplificado o processo que um usuário executa para gerar um código, porém, entender como a DSL funciona é também intrinsecamente importante. A linguagem foi montada a partir de várias classes, cada qual é responsável por uma parte da definição de um AC. Observando o diagrama

de classes, representado pela figura 10 a seguir, é possível compreender a estrutura e as relações das classes da ferramenta.

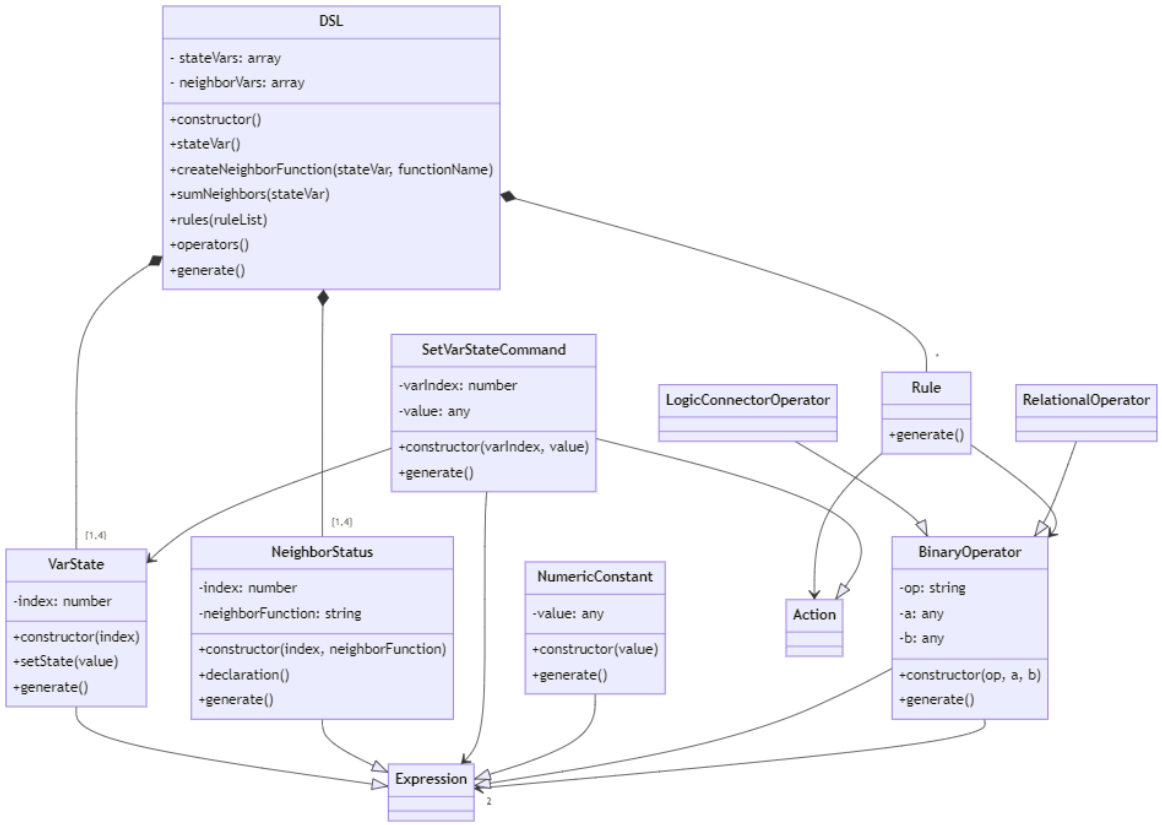

Figura 10: Diagrama de classes da DSL

A classe *DSL*, naturalmente, por ser a única que precisa ser instanciada pelo usuário, é a mais importante pois ela compõe as outras classes e é responsável também por guardar as funções de vizinhança, operadores e variáveis. Explicando suas funções de forma sucinta, o método *stateVar()* relaciona a classe *DSL* com a *StateVar* a medida que é através dela que são declaradas as variáveis de estado. Importante mencionar que o método *stateVar()* possui uma validação para não ser chamada mais de quatro vezes, respeitando dessa forma a limitação do shader e impedindo erros de usuário. Em seguida, *createNeighborFunction* é o método que recebe como parâmetro a variável estado definida pelo *stateVar()* (um index de valor igual a 1 à 4) e o nome da função de vizinhança desejada. O método *rules* é relativamente simples, ele apenas recebe a lista de regras inserida pelo usuário e instancia a classe *Rule*. Por último, *generate()*, método comum entre a maioria das classes do diagrama, é destacado pois é essa que irá exibir os trechos de código gerados a partir do *input* do usuário.

```
class DSL {
```

```
stateVar() {
    let index = this.stateVars.length;
    if(index < 4) {
        let sv = new VarState(index)
        this.stateVars.push(sv);
        return sv;
    } else {
        throw new Error("Muita variavel");
    }
}
createNeighborFunction(stateVar, functionName) {
    let index = stateVar.index;
    if(this.neighborVars[index]) {
        throw new Error("Vizinho ja usado");
    } else {
        let nv = new NeighborStatus(index, functionName);
        this.neighborVars[index] = nv;
        return nv;
    }
}
rules(ruleList) {
    this.ruleList = new RuleList(ruleList);
}
generate() {
    let resp = [];
    resp.push("vec4 executeRules(vec4 currentState) {");
    resp.push("\t/** Declarations **/");
    resp.push("\tvec4 nextState = vec4(0.0);");
    resp.push("\tvec4 neighborStatus= vec4(0.0);");
    resp.push("\n\t/** Neighbor Status **/");
    for(let n of this.neighborVars) {
        resp.push(n.declaration());
    }
    resp.push("\n\t/** Rules **/");
```

```
        resp.push(this.ruleList.generate());
        resp.push("\n\treturn nextState;");
        resp.push("}");
        return resp.join("\n");
    }
}
```

A classe *VarState* e *SetVarStateCommand* possuem uma associação entre si e trabalham sobre as variáveis de estado da célula. Na classe *VarState*, o método *setStatus(value)* cria uma nova instância da classe *SetVarStateCommand* passando o index (variável de estado) e a variável *value* (valor a ser atualizado da variável de estado). Em outras palavras a classe *VarState* representa o estado atual da célula (*currentState*), enquanto a *SetVarStateCommand*, ao ser chamada derivada de uma ação, atualiza esse valor atual para um novo valor (*nextState*).

```
class VarState {
    setState(value) {
        return new SetVarStateCommand(this.index, value);
    }
    generate() {
        return `currentState[${this.index}]`;
    }
}
class SetVarStateCommand {
    generate() {
        return `nextState[${this.varIndex}]=${this.value.generate()};`;
    }
}
```

NeighborStatus é uma classe que apenas será responsável por gerar código basicamente. Pelo método *declaration()* e *generate()*, a classe retorna as partes de código com a declaração de uma constante igualada a função escolhida pelo usuário e o status da vizinhança para aquela variável de estado. É importante saber quais as funções de vizinhança que podem ser usadas nesse método, visto que elas são estáticas no método por enquanto.

```
class NeighborStatus {
```

```
    declaration() {
        return
            `\tneighborStatus[${this.index}]=${this.neighborFunction}(${this.index});`;
    }
    generate() {
        return `neighborStatus[${this.index}]`;
    }
}
```

A classe *BinaryOperator* também não possui uma lógica muito complexa em seus métodos. Utilizando-se de conectores lógicos booleanos e operadores relacionais, essa classe retorna com o método *generate()* as expressões lógicas entre duas variáveis (a e b) com um operador lógico entre elas (op).

```
class BinaryOperator {
    generate() {
        return `(${this.a.generate()}${this.op}${this.b.generate()})`;
    }
}
```

Recebendo a lista de regras como parâmetro, a classe *Rule* irá utilizar as condições e ações definidas pelo usuário para montar blocos de lógica condicional. Como mencionado anteriormente, tal lista de regra é um *array* de dois elementos, condição e ação. Esses elementos são objetos das classes *BinaryOperator* e *SetVarStateCommand* respectivamente, seus métodos compõe a construção do trecho código da *Rule*. Basta apenas ao método *generate()* um laço, para percorrer toda a lista, buscar os trechos relativos à condição da *BinaryOperator* e ação da *SetVarStateCommand* (mudanças de estado), gerar um novo *array* e retorná-lo.

```
class RuleList {
    generate() {
        let resp = [];
        for(let r of this.list) {
            let [ condicao, acao ] = r;
            resp.push(`\tif(${condicao.generate()}){${acao.generate()}}`);
        }
```

```
        return resp.join("\n");
    }
}
```

Capítulo 6

Conclusão e Trabalhos Futuros

Nesse artigo, foi proposta uma DSL capaz de abstrair informações de uma simulação de autômato celular em um programa, o qual por sua vez irá gerar um código que virá a ser executado em placa gráfica.

Em termos de implementação, apenas foi contemplado WebGl (GLSL, shaders), porém, para planos futuros, a DSL terá suporte ao WebGPU, por ser uma tecnologia mais moderna e eficiente. Com mais tempo, seria interessante inserção de operadores aritméticos como adição, subtração, multiplicação e divisão para comportamentos mais elaborados. É possível também expandir essa ferramenta, adicionando conceitos como convolução e testar a eficácia da ferramenta para outros modelos de simulações que usam AC.

Além disso, a ferramenta tem muita margem de inovação no ramo de pesquisa e acadêmico. Pesquisadores e desenvolvedores, além de abstraírem seus próprios autômatos, podem utilizar essa DSL para aprofundamento nos conhecimentos de AC, shader, entre outros. Pode também ser utilizada como método de ensino de autômatos e, como gera um código que permite visualizar a evolução do AC, será ótimo para o aprendizado.

Como esse trabalho possui a base teórica (seção 2) é forte, bons exemplos para inspiração (seção 4) e um bom desenvolvimento para a ferramenta (seção 5), há confiança de que este projeto seja, de fato, bem sucedido. Espera-se, dessa forma, que a DSL desenvolvida seja útil para programadores não experientes em computação gráfica ou até como ponto de partida para trabalhos futuros.

Referências Bibliográficas

Gianpiero Cattaneo, Alberto Dennunzio, and Fabio Farina. A Full Cellular Automaton to Simulate Predator-Prey Systems. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Samira El Yacoubi, Bastien Chopard, and Stefania Bandini, editors, *Cellular Automata*, volume 4173, pages 446–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-40929-8 978-3-540-40932-8. doi: 10.1007/11861201_52. URL http://link.springer.com/10.1007/11861201_52. Series Title: Lecture Notes in Computer Science. 7

Q. Chen. Cellular automata. In S. E. Jørgensen, T. S. Chon, and F. Recknagel, editors, *WIT Transactions on State of the Art in Science and Engineering*, volume 1, pages 283–306. WIT Press, 1 edition, January 2009. ISBN 978-1-84564-207-5. doi: 10.2495/978-1-84564-207-5/16. URL http://library.witpress.com/viewpaper.asp?pcode=9781845642075-016-1. 1

Chronos. Shadertoy - Introduction to Game of Life, 2016. URL https://www.shadertoy.com/view/MtdXRn. 22

Fabio Farina and Alberto Dennunzio. A Predator-Prey Cellular Automaton with Parasitic Interactions and Environmental Effects. 2008. 6, 7

Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley, Upper Saddle River, NJ, 2011. ISBN 978-0-321-71294-3. 2, 14

M Gardner. Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life- M. Gardner - 1970. page 6, 1970. 4

Clemens Grelck, Frank Penczek, and Kai Trojahner. CAOS: A Domain-Specific Language for the Parallel Simulation of Cellular Automata. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 4671, pages 410–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73939-5 978-3-540-73940-1. doi: 10.1007/978-3-540-73940-1_41. URL http://link.springer.com/10.1007/978-3-540-73940-1_41. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science. 17, 18

Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. 2009. 2, 14, 15

Jörn Loviscach. Shader Programming: An Introduction Using the Effect Framework. *Revista de Informática Teórica e Aplicada*, 11(1):125–164, June 2004. ISSN 2175-2745. doi: 10.22456/2175-2745.5964. URL https://seer.ufrgs.br/index.php/rita/article/view/rita_v11_n1_p125-164. Number: 1. 16

Miles Berry. Conway's Game of Life in Processing, February 2020. URL https://www.youtube.com/watch?v=gu6SET230Yo. 4

Taleb A S Obaid. The Predator-Prey Model Simulation. 2013. 6, 7

Gonzales Patricio and Jen Lowe. The Book of Shaders, 2015. URL https://thebookofshaders.com/. 1, 15

Frank Penczek. On the Implementation of a Domain-Specific Language for the Parallel Simulation of Cellular Automata on Distributed Memory Parallel Computers, 2006. URL https://staff.fnwi.uva.nl/c.u.grelck/caos/Penczek06stud.pdf. 17

Aruna Raja and Devika Lakshmanan. Domain Specific Languages. *International Journal of Computer Applications*, 1(21), 2010. 15

Palash Sarkar. A brief history of cellular automata. *ACM Computing Surveys*, 32(1):80–107, March 2000. ISSN 0360-0300, 1557-7341. doi: 10.1145/349194.349202. URL https://dl.acm.org/doi/10.1145/349194.349202. 1, 3

ShadertoyBETA. Shadertoy BETA. URL https://www.shadertoy.com/. 22

Daniel Shiffman. The Nature of Code, 2012. URL https://natureofcode.com/book/chapter-7-cellular-automata/. 3

Oliver Ullrich and Daniel Lückerath. Modeling and Simulation Using Cellular Automata. *SNE Simulation Notes Europe*, 30(3):125–130, September 2020. ISSN 23059974, 23060271. doi: 10.11128/sne.30.en.10526. URL https://www.sne-journal.org/10526. 3, 4

Alex Valente. Predator and Prey, December 2018. URL https://github.com/AlexMFV/Predator-and-Prey/blob/210bfe72cf63c1bf27ba7f0be72fddd1ff430340/Predator_Prey/Predator_Prey.pde. original-date: 2018-12-13T21:47:28Z. 7

David Wolff. *OpenGL 4.0 Shading Language cookbook: over 60 highly focused, practical recipes to maximize your use of the OpenGL Shading Language*. Quick answers to common problems. Packt Publishing, Birmingham, 1st. ed edition, 2011. ISBN 978-1-84951-476-7. 1, 15, 16

Chao Zhang and Hessam S. Sarjoughian. Cellular Automata DEVS: A Modeling, Simulation, and Visualization Environment. In *Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques*, pages 11–19, Hong Kong China, September 2017. ACM. ISBN 978-1-4503-6388-4. doi: 10.1145/3173519.3173534. URL https://dl.acm.org/doi/10.1145/3173519.3173534. 6