Pontifícia Universidade Católica
do Rio de Janeiro

Pedro Boechat de Almeida Germano

Geração de Malhas Rodoviárias na GPU

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Alberto Barbosa Raposo

Rio de Janeiro

Março de 2014

Pedro Boechat de Almeida Germano

Geração de Malhas Rodoviárias na GPU

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Alberto Barbosa Raposo
Orientador
Departamento de Informática — PUC-Rio

Prof. Waldemar Celes Filho
Departamento de Informática — PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática — PUC-Rio

Prof. José Eugênio Leal
Coordenador(a) Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 28 de Março de 2014

Pedro Boechat de Almeida Germano

Graduou-se em Ciência da Computação pelo Centro Universitário da Cidade e continuou seus estudos no programa de Mestrado em Informática pela Pontifícia Universidade Católica do Rio de Janeiro. Possui 10 anos de experiência no mercado de desenvolvimento de software corporativo e, no momento, atua em projetos de visualização 3D no laboratório de computação gráfica Tecgraf.

Ficha Catalográfica

Germano, Pedro Boechat de Almeida

Geração de Malhas Rodoviárias na GPU / Pedro Boechat de Almeida Germano; orientador: Alberto Raposo – Rio de Janeiro: PUC, Departamento de Informática, 2014.

94 f.; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Modelagem Procedural. 3. Programação em placas gráficas. 4. Urbanismo. I. Alberto Raposo. II.Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Em primeiro lugar a Deus, pela Sua condução e providência.

Ao meu orientador Prof. Alberto Raposo, pela disponibilidade com a qual me orientou durante o mestrado.

Ao Prof. Luciano Pereira Soares, pelos conselhos valiosos.

Ao Tecgraf e aos colegas do Tecgraf: Daniel Ribeiro Trindade, Lucas Teixera, Manuel Loaiza, Paula Ceccon Ribeiro e Peter Furtado Dam.

À Universidade de Tecnologia de Graz e aos colegas do Instituto de Computação Gráfica e Visão Computacional: Prof. Dieter Schmalstieg, Jaime Garcia Guevara, Markus Steinberger, Michael Kenzel, Philip Voglreiter e Stefan Hauswiesner.

À minha namorada Priscilla, à minha família e aos meus amigos, pelo apoio e paciência incondicionais.

E, finalmente, à PUC–Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Resumo

Germano, Pedro Boechat de Almeida; Raposo, Alberto Barbosa. **Geração de Malhas Rodoviárias na GPU**. Rio de Janeiro, 2014. 94p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O primeiro estágio na linha de produção de um sistema de geração procedural de cidades é, tipicamente, a geração da malha rodoviária. Este trabalho apresenta um algoritmo para a geração de malhas rodoviárias em paralelo na GPU usando um modelo de execução baseado em filas de trabalho. Esse algoritmo recebe parâmetros declarativos, juntamente com mapas geográficos e sócio estatísticos, e produz uma representação em alto nível de uma malha rodoviária urbana.

Palavras-chave

Modelagem Procedural; Programação em Placas Gráficas; Geração de Geometria na GPU.

Abstract

Germano, Pedro Boechat de Almeida; Raposo, Alberto Barbosa (Advisor). **Road Network Generation on the GPU**. Rio de Janeiro, 2014. 94p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The first stage in the pipeline of a procedural city generation system is typically the generation of the road network. This work presents a parallel algorithm for road networks generation on the GPU, using a work-queue based execution model. This algorithm receives declarative parameters along with geographic and socio-statistical maps and produces a high level representation of an urban road network.

Keywords

Procedural Modeling; GPU Programming; Geometry Synthesis on the GPU.

Sumário

Lista de Figuras

1
Introdução

Um dos aspectos mais importantes na criação de aplicativos de realidade virtual é a criação do próprio mundo virtual. Na abordagem convencional, essa laboriosa tarefa é desempenhada por uma equipe de artistas 3d e, geralmente, toma boa parte do tempo do projeto. No entanto, muitos ambientes virtuais contém estruturas repetitivas e auto similares que poderiam ser modeladas muito mais rapidamente através do emprego de técnicas de geração procedural (Figura 1.1).

Figura 1.1 Exemplo de estrutura repetitiva e auto similar: plataforma de petróleo (representação artística) [20]

Técnicas de geração procedural têm sido desenvolvidas por mais de 20 anos. Entre as técnicas mais conhecidas estão a geração de texturas de materiais naturais [26], a geração de modelos de plantas e árvores [15], e a geração de terrenos [24]. Através do emprego dessas técnicas, ambientes complexos, que normalmente levariam meses para serem manualmente construídos, podem ser modelados em questão de segundos.

A essência das técnicas de geração procedural é descrever um recurso (geometria, imagem ou efeito) em termos de uma sequência de instruções. Uma mesma técnica pode ser empregada para produzir uma enorme variedade de recursos. O simples uso de geradores de números pseudoaleatórios é suficiente para produzir incontáveis texturas ou malhas de terrenos visualmente convincentes. Algoritmos recursivos, como fractais e L-Systems, podem gerar modelos que se assemelham a estruturas orgânicas encontradas na natureza, como flocos de neve e árvores.

A maioria das técnicas de geração procedural almeja reproduzir fenômenos naturais, mas muitas delas têm aplicabilidade, também, na reprodução de fenômenos humanos artificiais, como uma cidade.

1.1. Geração Procedural de Cidades

Modelar proceduralmente uma cidade não é algo trivial. Cidades não só são ambientes visualmente complexos, mas resultam de um elaborado processo evolucionário influenciado por muitos fatores.

Diversos estudos dividem conceitualmente uma cidade em três componentes perceptuais: malhas rodoviárias, vizinhanças e prédios [1][6]. Apoiado nessa simplificação é possível vislumbrar um processo onde cada um desses componentes seja gerado sequencialmente compondo o modelo maior de uma cidade (Figura 1.2).

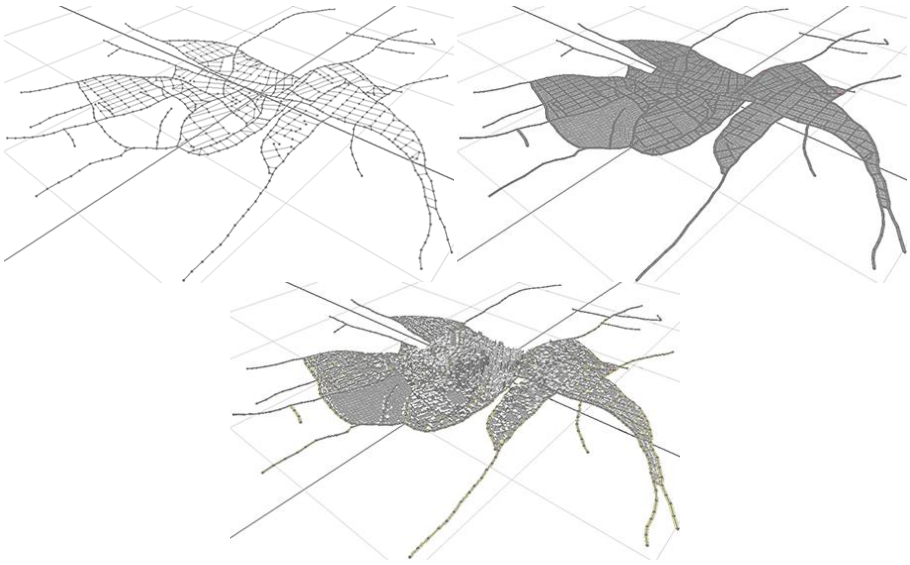

Figura 1.2 Linha de produção da geração de cidades: malha rodoviária (topo e esquerda), vizinhanças (topo e direita) e construções (abaixo)

De uma perspectiva visual, a malha rodoviária é o elemento mais impactante na estrutura de uma cidade. Lynch et al. [19] ressaltam que as rodovias são o meio pelo qual percebemos uma cidade e que, portanto, são responsáveis por estabelecer a mais forte impressão que temos dela. De uma perspectiva funcional, a malha rodoviária é extremamente significativa para uma cidade, pois consiste no meio pelo qual os indivíduos transitam por ela.

Fuesser [10] divide a malha rodoviária em duas categorias: primária e secundária. A malha primária, composta por vias principais, é responsável por conectar as áreas mais densamente populosas de uma cidade (Figura 1.3a). A malha secundária, composta por vias locais, é responsável por delinear as vizinhanças onde as construções da cidade serão erigidas (Figura 1.3b).

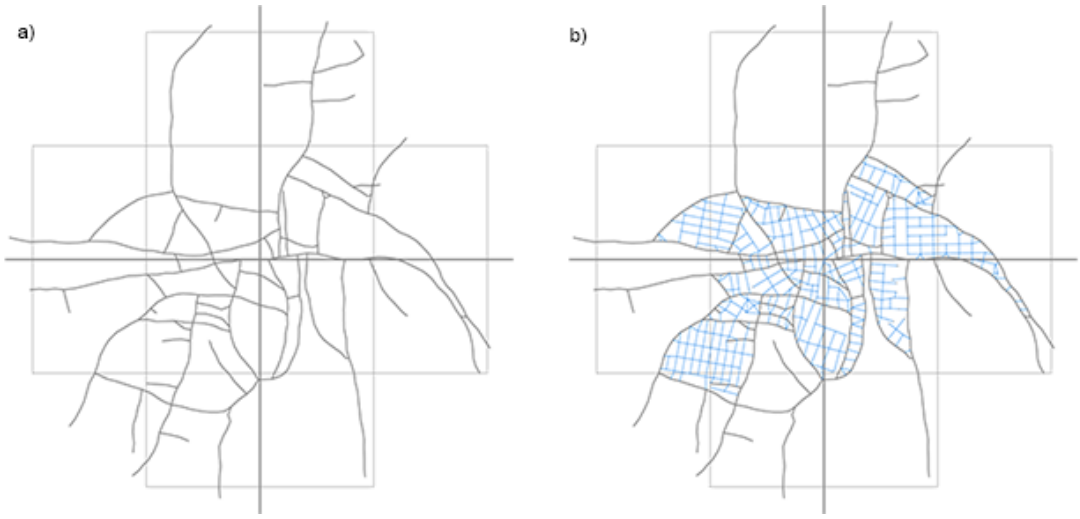

Figura 1.3 Categorias de malha rodoviária: (a) primária e (b) secundária

No capítulo 2 serão discutidas as técnicas de geração procedural mais comumente usadas para criar malhas rodoviárias por computador.

1.2. Objetivo

Este trabalho apresenta um algoritmo para a geração procedural de malhas rodoviárias na GPU. Com a execução na GPU, permitiu-se a geração de malhas rodoviárias massivas (com grande número de vias) com melhor

desempenho que na CPU. Até onde sabe o autor, não há outro trabalho que explore sua geração em multiprocessadores gráficos.

O algoritmo proposto usa mapas de imagem e parâmetros declarativos para produzir, de maneira semi-automatizada, uma representação em alto nível da malha rodoviária de uma cidade. A despeito da melhoria de desempenho, o algoritmo desenvolvido ainda não pode ser usado como parte integrante da linha de produção de um sistema completo de geração procedural de cidades em tempo real.

1.3. Estrutura do Documento

Este trabalho está estruturado da seguinte forma: o presente capítulo apresenta uma introdução ao assunto e os objetivos do trabalho. O segundo capítulo apresenta uma revisão das técnicas mais comuns na geração procedural de malhas rodoviárias. O terceiro capítulo apresenta o algoritmo proposto e detalha as características mais importantes da sua implementação na GPU. O quarto capítulo analisa a execução do algoritmo proposto na GPU sob diversos aspectos, comparando-a com a execução do mesmo algoritmo na CPU. O último capítulo apresenta conclusões, sugerindo possíveis trabalhos futuros que podem ser desenvolvidos a partir deste estudo.

2
Técnicas para Geração Procedural de Malhas Rodoviárias

No capítulo anterior consideraram-se aspectos gerais do processo de modelagem procedural. Delineou-se, também, uma visão do processo de geração procedural de cidades como uma linha de produção, da qual a geração da malha rodoviária faz parte como primeiro estágio. Neste capítulo, abordar-se-ão algumas técnicas comumente empregadas para a geração procedural de malhas rodoviárias.

2.1.
Geração Baseada em L-Systems

L-Systems são sistemas de reescrita paralela de cadeias de símbolos. Foram desenvolvidos em 1968 por Aristid Lindenmayer para descrever matematicamente o crescimento de plantas. L-Systems são similares às gramáticas de Chomsky [5], que operam sobre símbolos abstratos, ao invés de caracteres, e promovem a reescrita em paralelo, ao invés de serialmente.

Um L-System é caracterizado pelo terceto $G = (V, \omega, P)$ onde V é o conjunto dos símbolos (*alfabeto*), ω é a cadeia inicial de símbolos (*axioma*),e P é o conjunto de regras de produção. O processo de reescrita de símbolos ocorre através da aplicação sucessiva de regras de produção, na forma:

antecessor→sucessor

Onde antecessor e sucessor são cadeias de símbolos contidas no alfabeto.

A uma iteração do processo de reescrita se dá o nome de derivação. Uma derivação consiste na substituição de uma sub-cadeia da cadeia inicial que equivale à cadeia antecessora de uma regra de produção, pela cadeia sucessora da mesma regra (Figura 2.1).

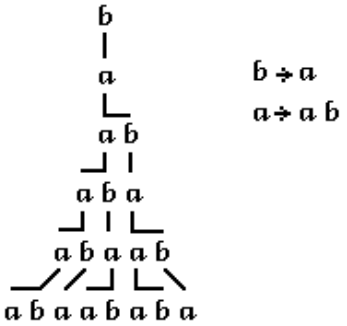

Figura 2.1 Exemplo de funcionamento do mecanismo de reescrita de um L-System [15]

Aristid et al. [15] usam o sistema Logo™ para interpretar as cadeias de símbolo que resultam da aplicação sucessiva de múltiplas derivações. O sistema Logo™, conhecido pelo uso de uma *tartaruga gráfica*, interpreta caracteres especiais (ex.: +, - e F) como instruções de desenho. Com a especificação do número de derivações (n) e do ângulo de rotação da *tartaruga gráfica* (δ), as cadeias de símbolos podem ser visualizadas como figuras 2d. Através dessas figuras é possível constatar a capacidade dos L-Systems de representar curvas e fractais com uma configuração simples do terceto (V, ω, P). Esse é o caso da curva quadrática de Koch (Figura 2.2a) e do triângulo de Sierpinski (Figura 2.2b).

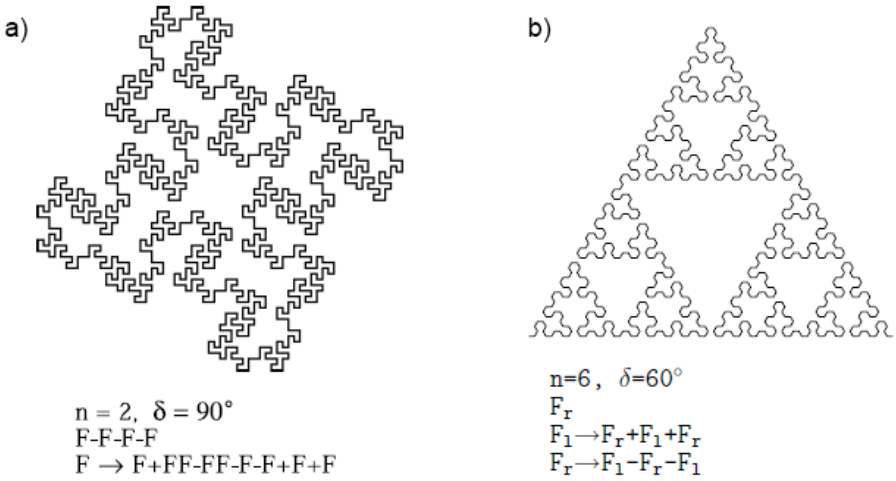

Figura 2.2 Representação de figuras matemáticas conhecidas por L-Systems: curva quadrática de Koch (a) e triângulo de Sierpinski (b) [15]

Aristid et al. também adicionam outras características aos L-Systems, como caracteres de ramificação (Figura 2.3a), regras estocásticas (Figura 2.3b) e símbolos parametrizáveis (Figura 2.3c). Isso não só permitiu simulações botânicas mais acuradas, mas aumentou o seu poder de representatividade dos L-Systems.

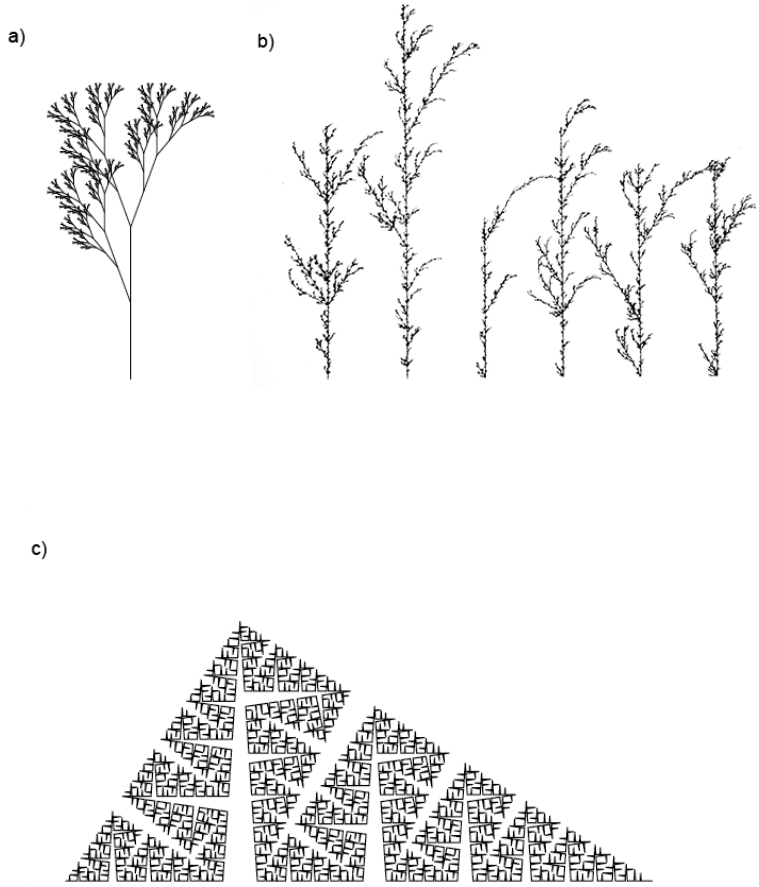

Figura 2.3 Exemplos de características dos L-Systems: ramificações (a), regras estocásticas (b) e parametrizações (c) [15]

Através dos anos, os L-Systems têm sido usados para atender aos mais variados propósitos, da geração de terrenos [2] à composição de músicas [21]. Concordante a isso, Parish et. al [11][25] apresentam um L-System capaz de gerar não só malhas rodoviárias mas prédios (Figura 2.4).

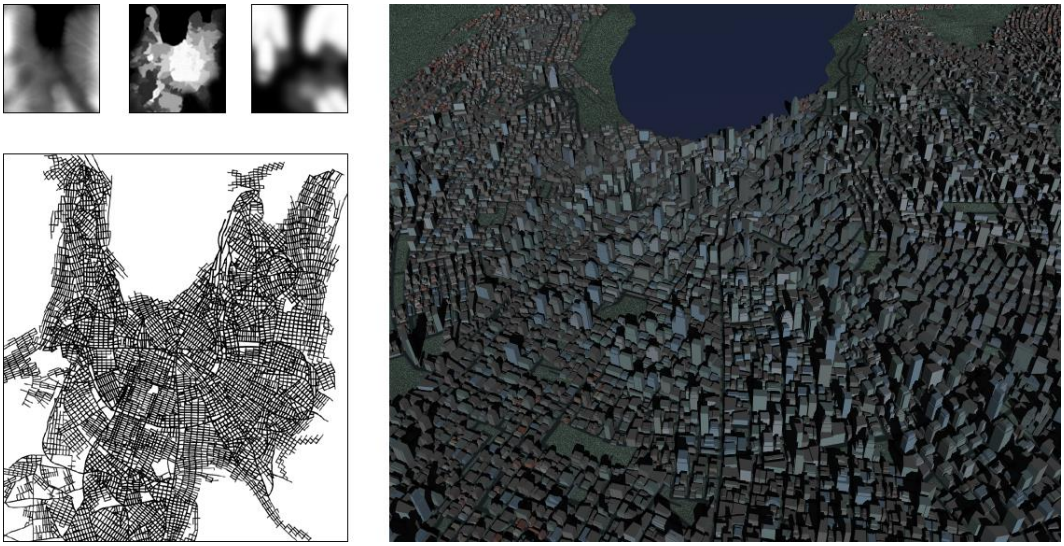

Figura 2.4 Malha rodoviária (esquerda) e prédios (direita) gerados através de L-Systems [25]

A geração de malhas rodoviárias proposta por esse trabalho considera dois tipos de rodovias: avenidas e ruas. Avenidas são vias arteriais que conectam áreas de alta densidade demográfica e ruas são vias capilares que provêem acesso às avenidas.

Apesar de adotar os L-Systems como técnica de geração, Parish et al. categorizam-nos, em sua forma original, como inadequados. Aponta-se suas limitações em adicionar novos comportamentos ao sistema e incapacidade de gerar estruturas que se assemelhem a malhas. Por isso, propõe-se um L-System com novas funcionalidades, chamado de L-System estendido.

Visando resolvera primeira inadequação, o L-System usa um conjunto fixo de regras de produção, chamadas de regras *template*. Usa, também, funções externas para computar o valor dos parâmetros dos símbolos sucessores das regras *template*. Com um conjunto fixo de regras de produção, a derivação fica responsável apenas por parte do comportamento da geração, sendo outra parte responsabilidade das funções externas. Assim, novos comportamentos podem ser adicionados através da configuração das funções externas, ao invés da criação de novas regras de produção.

A segunda inadequação é endereçada através da implementação de um mecanismo de aderência entre rodovias, chamado de auto sensibilidade. Esse mecanismo atua sobre o resultado da geração de forma a reforçar uma estrutura de malha no lugar da estrutura ramificada, típica dos L-Systems (Figura 2.5).

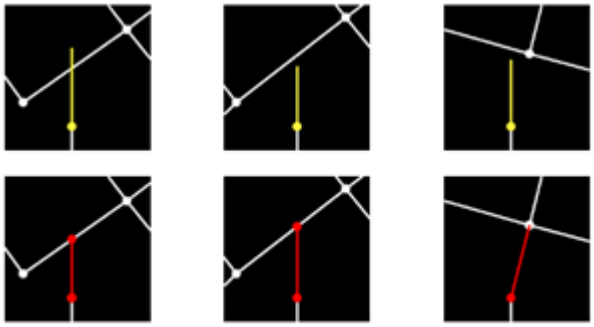

Figura 2.5 Os três casos considerados pela auto sensibilidade: cruzamento, aderência a aresta e aderência a vértice [11]

Conforme mencionado, quando uma regra *template* é aplicada a uma cadeia de símbolos, invocam-se funções externas para determinar o valor inicial dos parâmetros da cadeia de símbolos sucessores, chamada de *ideal successor*, desta regra (Figura 2.6).

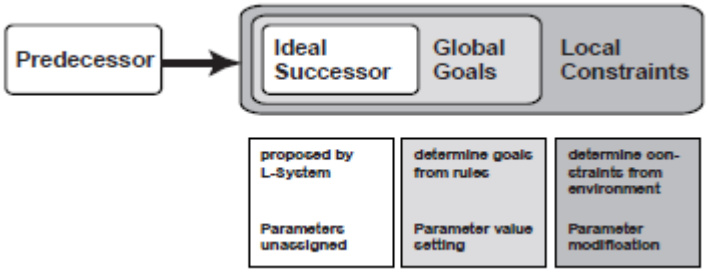

Figura 2.6 Fluxograma da aplicação de uma regra *template* a uma cadeia de símbolos [25]

A Figura 2.6 mostra que as funções externas são chamadas de maneira encadeada e são conhecidas por “objetivos globais” (*global goals*) e “restrições locais” (*local constraints*).

A função “objetivos globais*”* propõe valores iniciais para os parâmetros dos símbolos através da análise de regras urbanísticas. Existem, basicamente, duas regras: avenidas seguem a maior concentração de população e ruas obedecem a um padrão urbanístico. Ambas as regras são reforçadas a partir da consultas aos mapas de imagens informados como entrada para o sistema (Figura 2.7).

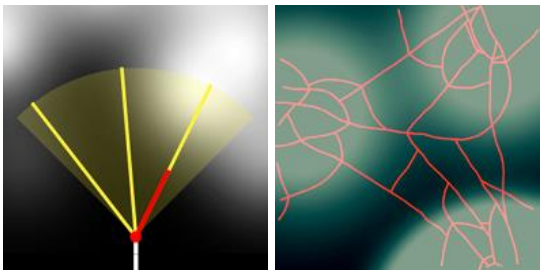

Figura 2.7 Avenidas seguem a maior concentração de população através de consultas aos mapas de imagens [11]

Já a função *restrições locais* pode ajustar os parâmetros propostos pela função anterior caso encontre irregularidades no posicionamento de uma rodovia, como quando se encontram sobre um corpo d'água. Essa avaliação consiste na consulta a mapas de imagens que retratam as características da região onde a malha rodoviária é gerada (Figura 2.8).

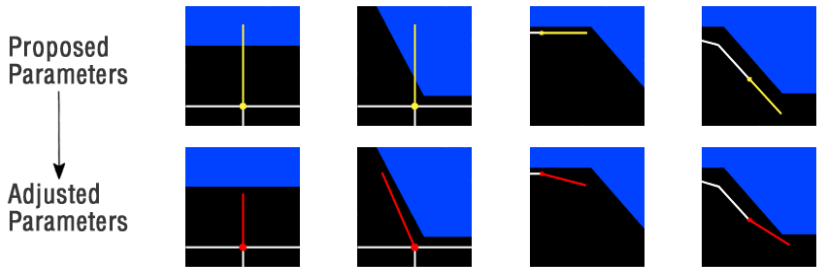

Figura 2.8 Ajuste da direção e orientação da via a fim de adequá-la aos mapas [11]

Comparando um mapa gerado com o mapa de Manhattan é possível constatar a semelhança bastante acentuada entre o resultado gerado e a malha real (Figura 2.9).

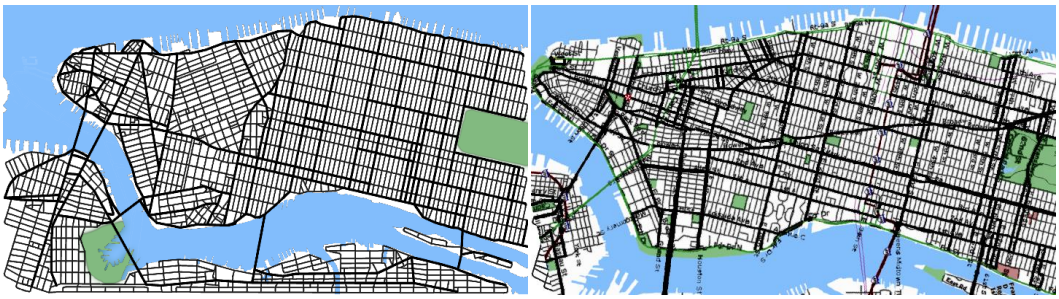

Figura 2.9 Mapa rodoviário gerado por L-Systems (esquerda) e mapa rodoviário real de Manhattan (direita) [25]

Reporta-se, ao fim do trabalho, que essa técnica representa um avanço em relação às existentes até o momento por não depender de fotografias aéreas e por poder gerar uma variedade infinita de cidades a partir do mesmo conjunto de entrada.

2.2. Geração Baseada em Modelos

Sun et al. [31] desenvolvem uma técnica baseada no L-System estendido de Parish (Seção 2.1). Baseado na ideia de que o uso de regras de produção dificulta a adição de novos comportamentos, sugere-se a aplicação de modelos (*templates*) de padrões urbanísticos. Argumenta-se que um padrão é apenas uma regularidade percebida no meio de uma diversidade de aparências e que, por isso, sua aplicação não deve levar em consideração detalhes ou exceções.

Nesta técnica, usa-se um modelo para cada um dos padrões urbanísticos mais comuns - natural (Figura 2.10a), de rasterização (Figura 2.10b), radial (Figura 2.10c) -e um para a mistura dos três (Figura 2.10d).

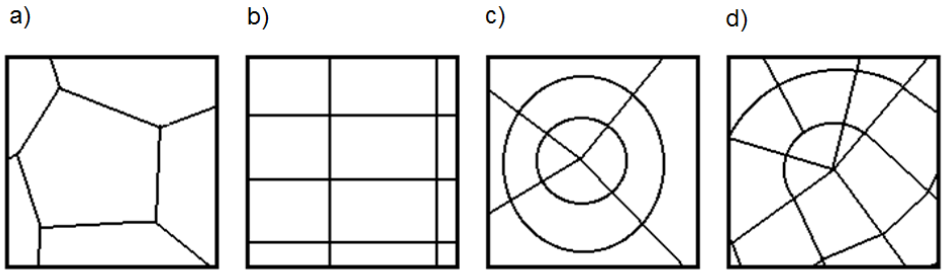

Figura 2.10 Padrões urbanísticos: (a) natural, (b) de rasterização, (c) radial e (d) misturado [31]

Como entrada, a técnica espera receber um mapa com informações geográficas (corpos d'água, terra e vegetações), um mapa de altura (*heightmap*) e um mapa especificando a densidade populacional da região onde será gerada a malha.

O processo de geração ocorre em seis etapas: a aplicação do modelo para as avenidas, validação das restrições, conexão de pontos, modificação das formas, extração de regiões e a aplicação do modelo para as ruas (Figura 2.11).

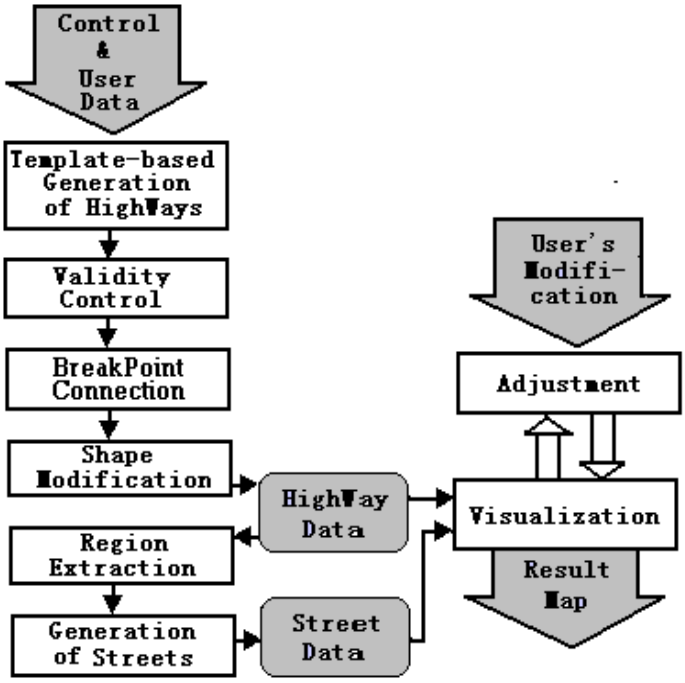

Figura 2.11 Fluxograma do processo de geração baseada em modelos [31]

A primeira etapa gera arestas, que representam avenidas, segundo o modelo escolhido. Para o modelo natural (*population-based template*), computa-se um diagrama de Voronoi usando os pontos de maior densidade do mapa. As arestas do diagrama são, então, usadas como avenidas. Para os outros modelos, aplica-se um algoritmo iterativo que gera pontos até que se alcance a caixa envolvente da região. Esses pontos são conectados par a par, de acordo com a sua localização, gerando as arestas.

A validação das restrições garante que avenidas não cruzem áreas ilegais. Dependendo da distância percorrida dentro de uma área ilegal, uma avenida pode ser ajustada diferentemente (Figura 2.12).

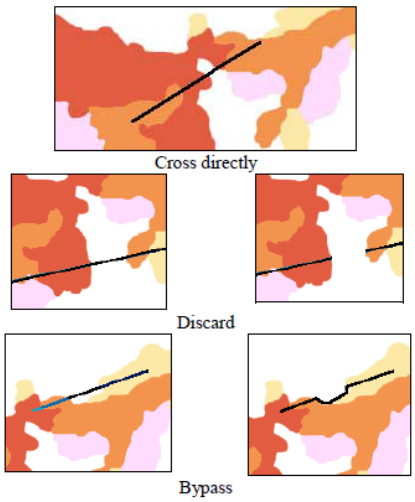

Figura 2.12 Avenidas ajustadas diferentemente pela validação das restrições [31]

Para reforçar o aspecto de malha, a etapa de conexão de pontos verifica a proximidade dos vértices das avenidas. Para cada aresta, consultam-se, com um arco de ângulo parametrizável, os vértices das arestas que podem ser conectadas a ela (Figura 2.13).

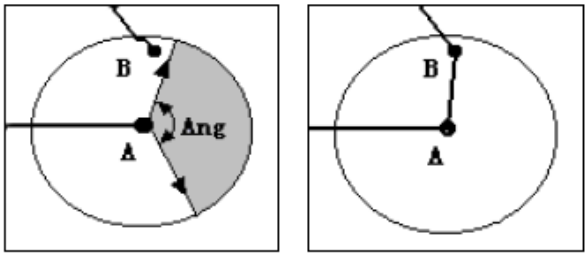

Figura 2.13 As arestas são conectadas às outras através de uma consulta em arco [31]

A conexão de pontos também trata o caso particular de avenidas interrompidas à beira mar. Nesses casos segue-se a linha costeira até que se encontre outra rodovia, conectando-as (Figura 2.14).

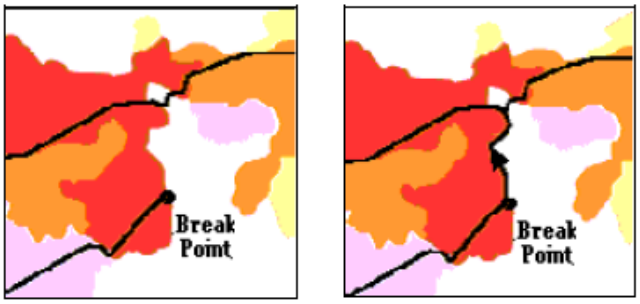

Figura 2.14 As vias interrompidas por corpos d'água seguem a linha costeira até se encontrarem com outra [31]

Por evitarem grandes alterações de altitude, rodovias geralmente possuem um aspecto curvilíneo. Para representar esse comportamento, a modificação de forma divide as arestas das rodovias em segmentos, orientando-os de acordo com o vetor gradiente da menor elevação (Figura 2.15).

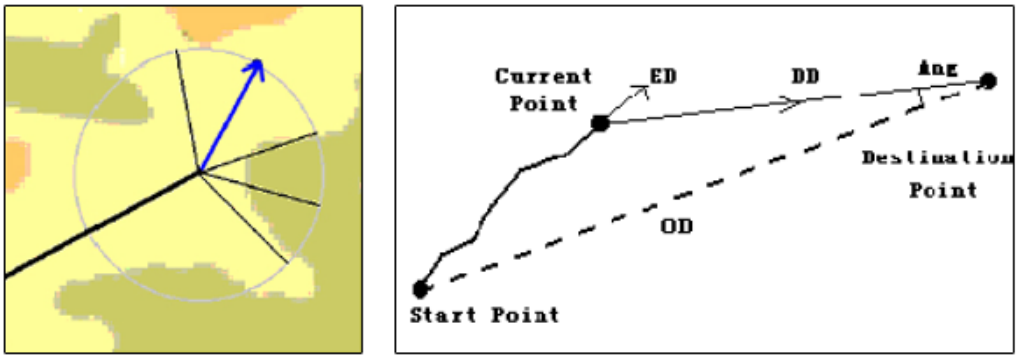

Figura 2.15 Arestas das rodovias são modificadas de acordo com os gradientes de elevação do terreno [31]

Após a geração das avenidas é possível identificar áreas chamadas de regiões de transporte local. Nas regiões de transporte local expandem-se as ruas a fim de delinear quarteirões e prover acesso às avenidas próximas. Para extrair essas áreas, sintetiza-se uma imagem a partir do mapa geográfico e da rasterização das avenidas (Figura 2.16a e b). Através do processamento dessa imagem identificam-se, então, as áreas que representam as regiões desejadas (Figura 2.16c).

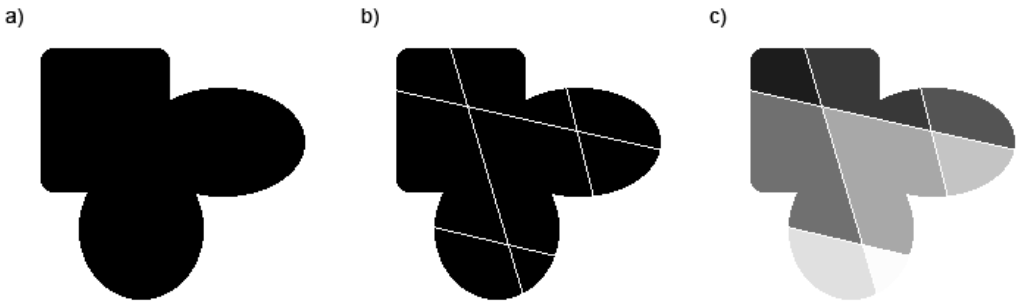

Figura 2.16 Sintetização de imagens para encontrar as regiões de transporte local: (a) mapa geográfico, (b) mapa geográfico + avenidas, (c) mapa geográfico + avenidas + identificação de regiões desconexas [31]

Como mencionado, dentro das regiões de transporte local, expandem-se as ruas. A expansão aplica o modelo de rasterização para gerar as arestas das ruas em cada região. As arestas das ruas podem terminar de três formas. A primeira forma (Figura 2.17a) constitui uma conexão válida entre duas ruas, a segunda (Figura 2.17b), uma conexão entre rua e avenida, e a terceira (Figura 2.17c) uma expansão dentro de uma área inválida. Na última forma, a aresta é removida e a rua desconsiderada.

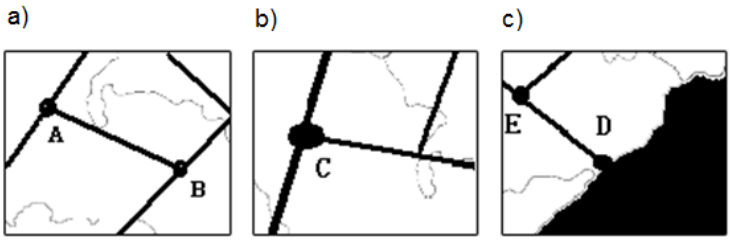

Figura 2.17 Três formas de arestas em uma região de transporte local: (a) conexão válida entre duas ruas, (c) conexão entre rua e avenida e (d) expansão dentro de uma área inválida [31]

A Figura 2.18 compara o mapa rodoviário da China com um mapa gerado mostra a proximidade entre uma malha rodoviária real e o resultado obtido.

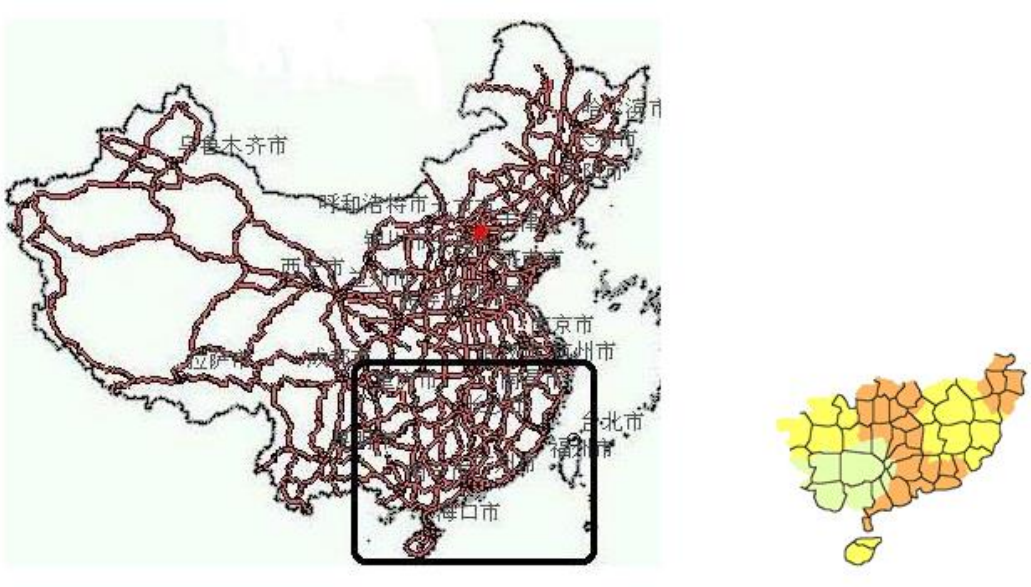

Figura 2.18 Mapa rodoviário real da China (esquerda) e mapa rodoviário gerado por modelos (direita) [31]

Conclui-se o trabalho apresentando algumas limitações que precisariam ser endereçadas. Por exemplo, não é possível usar essa técnica para representar malhas rodoviárias em evolução, onde existem longas rodovias conectando centros populacionais dispersos e não há regiões de transporte local os intermediando.

2.3. Geração Baseada em Campos Tensores

Essa técnica foi desenvolvida por Chen et al. [4] com o objetivo de facilitar a edição da malha rodoviária pelo usuário. Ela se baseia no fato de existirem duas direções dominantes em todos os padrões urbanísticos. Como campos tensores podem ser decompostos em dois conjuntos de *hyperstreamlines* - um seguindo o campo dos autovetores maiores e outro seguindo o campo de autovetores menores - é possível gerar malhas rodoviárias com padrões conhecidos através do uso de campos tensores. Além do mais, relacionando um campo tensor à malha, é possível refletir modificações feitas ao primeiro no segundo a Figura 2.19.

Figura 2.19 Alterações feitas ao campo tensor refletem diretamente na malha rodoviária gerada a partir dele [4]

O campo tensor usado na geração da malha é produzido pela combinação de quatro campos-bases individuais. Esses campos-bases são identificados como: gradeado, radial, dos limites e das alturas. Os dois primeiros campos são computados a partir das entradas do usuário. O primeiro descreve um padrão gradeado global (Figura 2.20a) e o segundo os padrões radiais locais (Figura 2.20b) da malha. O campo dos limites é computado a partir do processamento do mapa de corpos d'água. Ele extrai do mapa as poligonais (*polylines*) dos corpos d'água e cria campos tensores a partir delas (Figura 2.20c). O campo das alturas computado a partir do processamento do mapa de elevações. Extraem-se os gradientes das elevações e computa-se o campo tensor a partir deles.

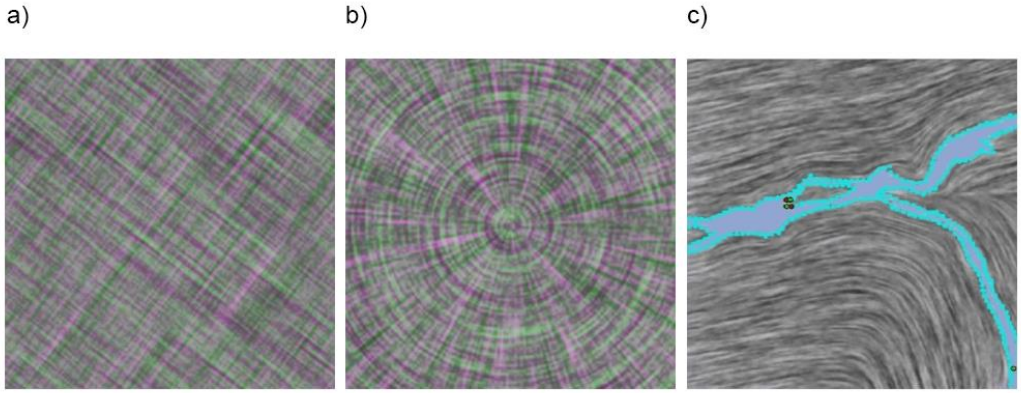

Figura 2.20 Campos tensores bases: (a) gradeado, (b) radial e (c) de corpos d'água [4]

Os campos base são combinados a partir de seu somatório, usando uma função radial como filtro.

A geração da malha é feita através de uma técnica denominada rastreio de *hyperstreamline*. Primeiramente, pontos iniciais para o rastreio são lidos da entrada ou gerados proceduralmente através de alguma função de ruídos. Cada um dos pontos dá início a um rastreio, que segue uma hyperstreamline gerando rodovias, representadas como arestas de um grafo, segundo a orientação dos autovetores do campo. O rastreio prevê a colisão entre a rodovia em expansão e rodovias existentes, bem como a aderência de rodovias próximas. Isso reforça o aspecto de malha do grafo que é gerado (Figura 2.21).

Figura 2.21 Malha rodoviária gerada sem (esquerda) e com (direita) aderência de vias próximas [4]

O trabalho apresenta muitas reproduções convincentes de malhas rodoviários reais, mas esclarece que o objetivo da técnica é gerar malhas rodoviárias inspiradas por malhas reais, mas não exatamente replicá-las (Figura 2.22).

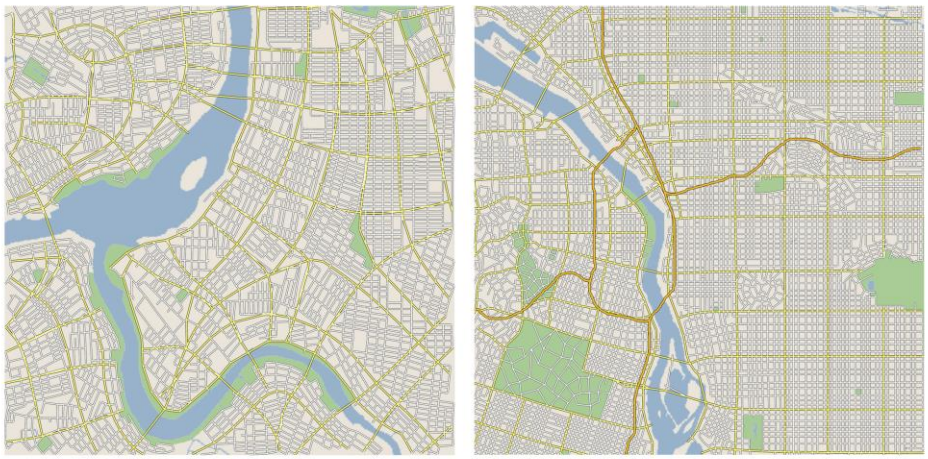

Figura 2.22 Mapas rodoviários gerados por campos tensores são bastante convincentes [4]

2.4. Discussão

As técnicas apresentadas estão em sequência cronológica e são, até certo ponto, inter-relacionadas. A geração baseada em L-Systems é a primeira técnica a possibilitar a geração de malhas rodoviárias sem o uso de fotografias aéreas. A geração baseada em modelos propõe uma alternativa às regras de produção, o que a torna fácil de ser estendida. A geração baseada em campos tensores permite ao usuário editar a malha rodoviária de maneira mais direta. O que elas possuem em comum é que todas provêm facilidades ao usuário final. Elas apresentam contribuições ao estudo da geração de cidades mas, por não apresentarem dados relativos ao desempenho, não podem ser avaliadas nesse sentido.

Como será argumentado nos capítulos seguintes, este trabalho se apoia sobre os anteriores, mas explora a execução em dispositivos massivamente paralelos. Para que se possam apurar as vantagens da execução paralela, é desejável compará-la à execução serial. Por não apresentarem dados relativos ao desempenho, não é possível usar os trabalhos anteriores para realizar essas comparações. Assim, faz-se necessária a proposição de um algoritmo que execute em ambas as plataformas.

3
Geração de Malhas Rodoviárias na GPU

No capítulo anterior analisaram-se algumas técnicas comumente empregadas na geração procedural de malhas rodoviárias. Neste capítulo apresentar-se-á um algoritmo para a geração procedural de malhas rodoviárias em paralelo na GPU. Primeiro, explicar-se-á este algoritmo em linhas gerais, para depois expor suas particularidades de implementação na GPU. O algoritmo será explicado de maneira agnóstica quanto à plataforma para que seja possível implementá-lo tanto na CPU quanto na GPU. Também é importante ressaltar que os detalhes expostos não compõem o quadro total da implementação na GPU. Expuseram-se apenas aqueles que estão intimamente relacionados aos resultados do Capítulo 4.

3.1.
Visão Global do Algoritmo

O algoritmo proposto por este trabalho se baseia no modelo da geração de malhas rodoviárias de Parish et al. [11][25]. Conceitualmente, esse modelo propõe que vias urbanas sejam geradas de maneira expansiva e em etapas, onde em cada etapa se avalia novos objetivos de expansão. Esse modelo também propõe que a expansão de uma via seja independente da expansão das outras, sendo interrompida apenas pela aplicação de regras restritivas relacionadas à região sob a malha. A escolha por esse modelo se justifica pela sua adequação à execução paralela, já que ele divide o algoritmo em partes e determina que essas partes sejam computadas de maneira independente (*data independency*).

A fim de garantir algum nível de controle sobre o resultado da geração, o algoritmo apresentado permite a configuração dos objetivos e regras restritivas mencionadas acima. Isso pode ser feito através da definição dos mapas de imagens e parâmetros declarativos, detalhados na Seção 3.1.1.

Enquanto o modelo de Parish et al. propõe a geração de ruas e avenidas simultaneamente, este algoritmo divide o *processo de geração* em três partes

(Figura 3.1). O modelo de Parish et al. também propõe a interrupção da expansão de ruas e avenidas que se cruzam ou se encontram muito próximas, o que não é considerado por este algoritmo. O motivo dessas adaptações será apresentado na Seção 3.1.2.

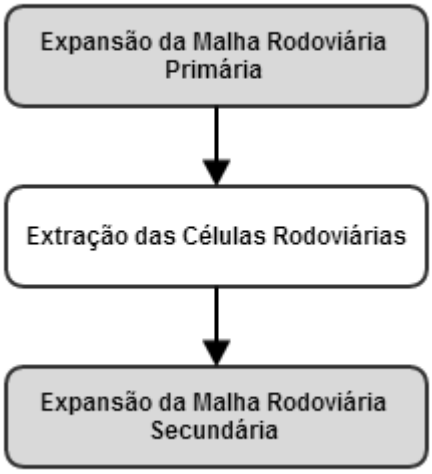

Figura 3.1 Fluxograma das partes do processo de geração

Apesar das naturais semelhanças com o L-System estendido de Parish et al., este algoritmo não usa um L-System para gerar malhas rodoviárias. Antes, apoia-se na *execução baseada em filas de trabalho*, cuja importância para a sua implementação na GPU será discutida na Seção3.1.3. É interessante notar que a correlação entre L-Systems e filas de trabalho fora feita anteriormente em [16][17]. Nesses trabalhos usam-se filas de trabalho para realizar a interpretação de L-Systems na GPU.

3.1.1. Mapas de Imagem e Parâmetros Declarativos

Os mapas de imagem são imagens 2d que descrevem dados geográficos e sócio estatísticos da região onde será gerada a malha rodoviária. Durante a simulação, esses dados são consultados para deduzir tanto os objetivos quanto as regras restritivas de uma via. São eles: mapa de corpos d'água (Figura 3.2a), densidade populacional (Figura 3.2b), zonas de bloqueio (Figura 3.2c) e padrões urbanísticos (Figura 3.2d).

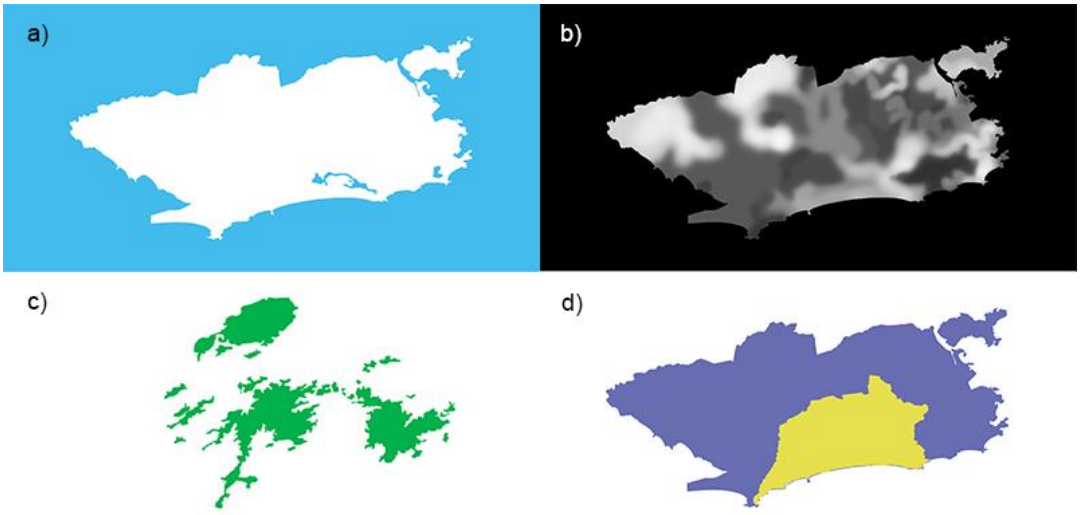

Figura 3.2 Os quatro mapas de imagem usados pelo algoritmo proposto: (a) mapa de corpos d'água, (b) mapa da densidade populacional, (c) mapa de zonas de bloqueio e (d) mapa de padrões urbanísticos

Note que os mapas podem ser provenientes de fontes reais, ou podem ser gerados a partir de simulações. Abordagens como [14] podem ser usadas para gerar mapas verossímeis.

Os parâmetros declarativos são pares "chave e valor" usados para configurar diferentes atributos da implementação desenvolvida. Alguns parâmetros atribuem grandeza às características da simulação, como, por exemplo, o tamanho das vias principais e locais. Outros regulam recursos de memória alocados antes do início da execução, como, por exemplo, o tamanho do vetor de vértices do grafo que representa as malhas. Ainda, outros parâmetros configuram características próprias da visualização, que não são relacionadas ao algoritmo de geração. Por exemplo, o tamanho, em pixels, de um vértice na tela.

No Apêndice A encontram-se discriminados todos os parâmetros usados na implementação desenvolvida e seus respectivos significados.

3.1.2. O Processo de Geração

O processo de geração tenta reproduzir a evolução da malha rodoviária através do tempo, avaliando, para cada via candidata, objetivos e regras

restritivas. Trata-se de uma simulação similar à baseada em agentes, onde o resultado não pode ser aferido através da resolução de uma equação, mas pela atualização, passo a passo, de entidades que possuem estados e comportamentos. Seguindo a analogia com a simulação de agentes, neste trabalho as entidades detêm informações de uma via candidata (estado) e decidem *como* expandi-la e *quando* adicioná-la à malha (comportamento).

Outra consideração importante sobre o processo é sua condição de parada. O modelo de Parish et al. sugere que a malha secundária reduza a concentração populacional ao se expandir para que as rodovias parem ao chegar a uma região inabitada. A redução (ou difusão) da concentração populacional, descrita por uma imagem bidimensional, é calculada através de uma equação diferencial parcial (EDP) pelo método das diferenças finitas [28]. Como o método das diferenças finitas representa a EDP como uma matriz trigonal de índices [27], atualizar esses índices a cada passo da expansão poderia gerar problemas de sincronismo na GPU. Por isso, neste trabalho, propõe-se a separação do processo de geração em três partes. Na primeira parte as vias principais se expandem segundo a descrição estática da concentração populacional. Essa expansão estabelece regiões fechadas, chamadas de *células rodoviárias* (Figura 3.3a), cuja identificação (ou extração) consiste na segunda parte. Na terceira parte as vias locais se expandem dentro dos limites das células rodoviárias encontradas na primeira fase (Figura 3.3b).

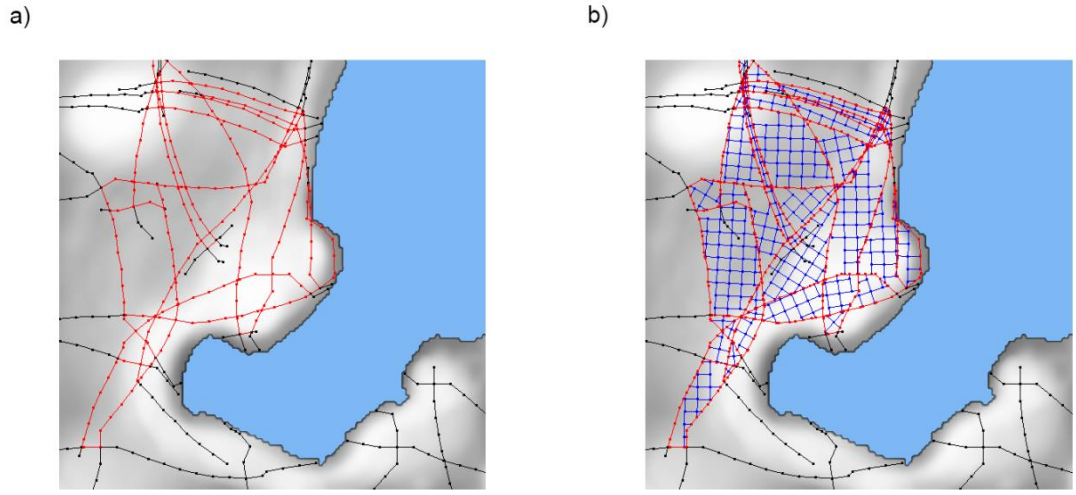

Figura 3.3 Ilustração das duas expansões do processo de geração: (a) expansão da malha rodoviária primária e (b) expansão da malha rodoviária secundária

Ainda sobre as condições de parada, o modelo de Parish et al. também sugere que a expansão de uma via seja interrompida quando ela aderir à outra (Figura 2.5). No entanto, é possível notar que a interrupção no caso de aderência cria uma dependência de ordem no algoritmo. Por exemplo, no caso do cruzamento das vias A e B (Figura 3.4a), a malha rodoviária gerada se A aderir à B (Figura 3.4b) será diferente da gerada se B aderir à A (Figura 3.4c). Para evitar essa dependência, propõe-se ignorar essa interrupção.

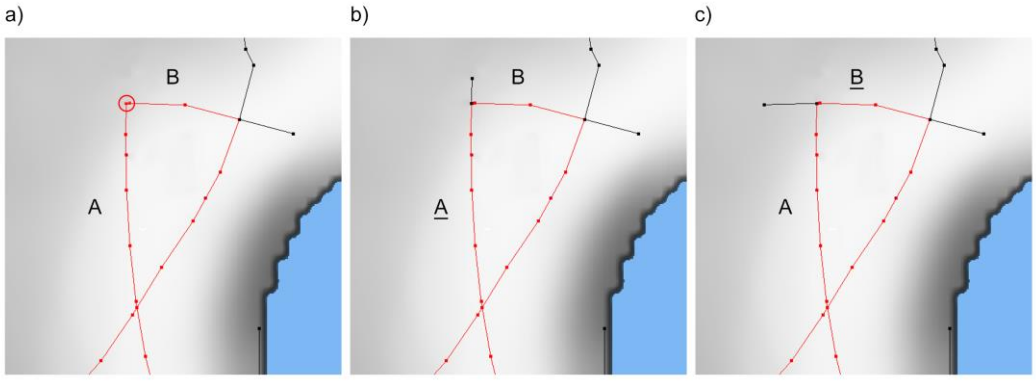

Figura 3.4 (a) As arestas A e B se cruzam. Se (b) A aderir a B, a malha rodoviária resultante será diferente do que se (c) B aderir a A

Em consequência da ausência de ambas as condições de parada (por densidade populacional e por aderência), as via principais podem expandir indefinidamente. Para que isso não ocorra, cada via principal expande apenas um número finito de vezes. Note que esse número não corresponde ao número de passos da simulação. O número de vezes de expansão de uma via determina exclusivamente o período de expansão da via em questão, enquanto o número de passos da simulação determina o período de expansão de todas as vias. O impacto visual decorrente dessa mudança no modelo de Parish et al. é explorado na Seção 4.4.

Assim, quanto às condições de parada, o algoritmo proposto adota diferentes condições para cada parte do processo. Na primeira, as vias principais se expandem por um número finito de vezes, na segunda, as células rodoviárias são extraídas até que não haja mais nenhuma, e na terceira as vias locais se expandem até colidir com o bordo das células rodoviárias. Adicionalmente, as expansões são limitadas por um número finito de passos.

Para representar as malhas rodoviárias no computador, escolheu-se um grafo cíclico não dirigido. As arestas são, naturalmente, vias candidatas que foram adicionadas à malha. A escolha por um grafo não dirigido deve-se ao fato de que sem direção não há duplicidades desnecessárias, dado que não podem existir duas arestas ligando o mesmo par de vértices. Além disso, para executar o algoritmo de extração de células rodoviárias, descrito na Seção 3.3, impediu-se o cruzamento entre arestas (*arestas sobrepostas*) criando-se um vértice nas interseções (Figura 3.5).

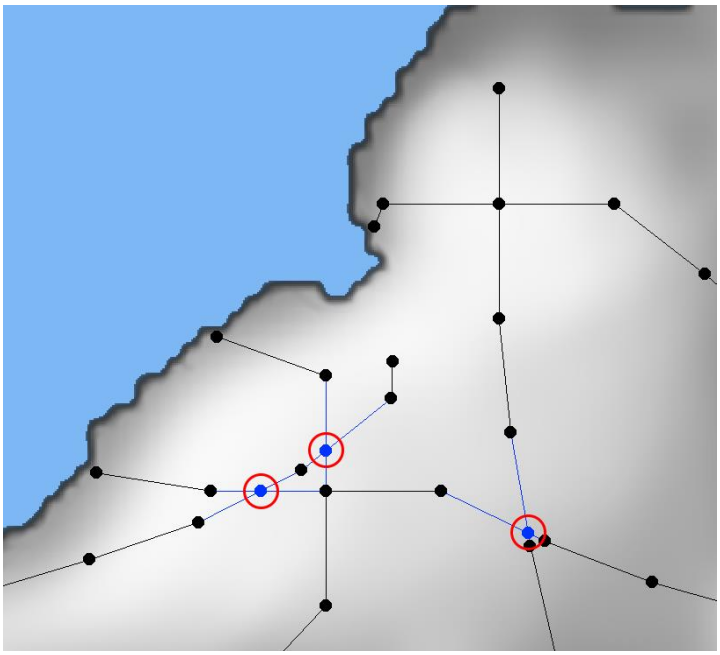

Figura 3.5 Arestas que se cruzam precisam de um vértice intercessor por causa da extração de células rodoviárias

3.1.3. Execução Baseada em Filas de Trabalho

Crê-se, intuitivamente, que se um algoritmo puder ser dividido em muitas partes, mais partes poderão ser executadas em paralelo e, assim, maior desempenho se obterá. No entanto, na prática, verificam-se alguns obstáculos à divisão de um problema em partes.

Quão mais divisível for um algoritmo, maior será o custo para assegurar o acesso concorrente aos dados compartilhados entre suas partes [3]. Mesmo sendo altamente divisível, nem todas as partes terão o mesmo tempo de processamento. Além disso, nem todos os algoritmos são facilmente divisíveis antes de serem executados, ou em outras palavras, há casos onde não é trivial antecipar o caminho de execução de um algoritmo sem que antes o executemos (*data-dependant execution path*) [30].

Assim, para tirar real proveito do paralelismo, um algoritmo deve lidar com o controle de acesso concorrente à dados (*sincronismo*), com os diferentes tempos de processamentos de suas partes (*heterogeneidade*) e com a imprevisibilidade na quantidade destas (*dinamismo*).

A execução baseada em filas de trabalho tenta facilitar a elaboração de estratégias que enderecemos aspectos mencionados acima. Filas de trabalho

são coleções de elementos, chamados *itens de trabalho*, que representam os dados necessários para executar uma parte de um algoritmo por um processo concorrente (*thread*). As filas de trabalho devem permitir a adição e remoção concorrente de elementos de maneira segura, através do uso de mecanismos de exclusão mútua (*blocking*) ou sem o uso de mecanismos de exclusão mútua (*non-blocking*). No geral, filas *non-blocking* apresentam melhor desempenho [3]. Elas podem adotar o comportamento padrão (*first in, first out*) ou ser ordenadas por algum critério (ex.: tempo de execução). Filas ordenadas servem como ponto inicial para a implementação do agendamento dinâmico da execução dos itens de trabalho [30]. Por fim, do ponto de vista de alocação de memória, as filas de trabalho podem ser estáticas ou dinâmicas. Filas dinâmicas recorrem a um mecanismo de realocação de memória *on demand*, enquanto filas estáticas geralmente em buffers de memória pré-alocada.

3.2. Expansão da Malha Rodoviária Primária

A expansão da malha rodoviária primária começa com a criação de quatro itens de trabalho para cada elemento definido no parâmetro de pontos iniciais da simulação (*spawn_points*). Esses itens de trabalho representam cada um, vias principais *candidatas*, como mencionado na Seção 3.1.2. No decorrer da expansão essas vias candidatas são avaliadas e, então, descartadas ou instanciadas (adicionadas à malha). Durante a instanciação, três novos itens de trabalho podem ser criados: duas ramificações e uma nova via candidata, representando a continuação da via originária. As ramificações também são avaliadas e, eventualmente, geram novas vias candidatas, retroalimentando, assim, o *ciclo de expansão* (Figura 3.6).

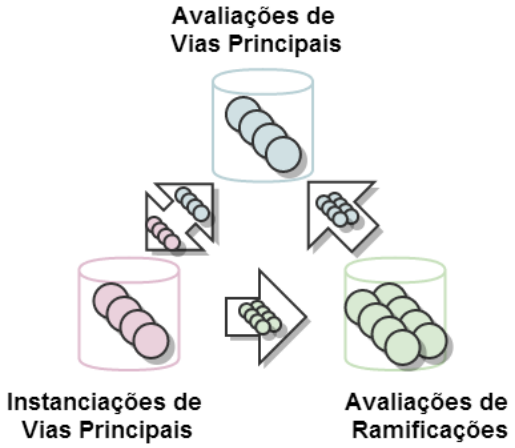

Figura 3.6 Derivação é uma iteração no ciclo de expansão

À cada iteração do ciclo de expansão dá-se o nome de derivação (em alusão a execução dos L-Systems). O ciclo de expansão realiza tantas iterações quantas definidas pelo parâmetro *num_highway_derivations*. Na execução do ciclo, usa-se duas listas de filas de trabalho de maneira similar ao uso de buffers gráficos duplos. Durante uma derivação, os itens de trabalho a ser executados são retirados das filas pertencentes à lista frontal, enquanto que os novos itens de trabalhos, criados durante a execução dos procedimentos, são adicionados às filas da lista traseira. Ao final da derivação, os papéis das listas, frontal e traseira, são alternados. Assim, garante-se que todos os itens de trabalho de uma derivação são processados antes que os itens de trabalho de outra derivação o sejam.

Como é possível observar, a expansão da malha rodoviária é composta por três partes (procedimentos) distintas: a avaliação das vias principais, a avaliação das ramificações e a instanciação das vias principais.

Na avaliação de uma via principal verifica-se se a via candidata está em conformidade com as regras de restrição. Regras de restrição são impeditivos à instanciação de uma via candidata em uma determinada posição. Por exemplo, uma via candidata não pode penetrar corpos d'água ou zonas bloqueadas. Para verificar se uma via candidata se encontra sob restrições locais, e eventualmente reposicioná-la, é executado o algoritmo de desvio de obstáculos (Figura 3.7).

Algorithm 1 Avalia Restrições Locais para uma Via Candidata

```
1:  function EVALUATE(road, map)
2:      length ← road.length
3:      while length >= min_road_length do
4:          angle ← road.angle
5:          while angle < max_obstacle_deviation_angle do
6:              if CASTRAY(map, road, angle, length) then
7:                  go to 13
8:              end if
9:              angle + +
10:         end while
11:         length − −
12:     end while
13:     road.angle ← angle
14:     road.length ← length
15: end function
```

Figura 3.7 Pseudocódigo do procedimento de avaliação de uma via candidata (*desvio de obstáculos*)

Esse algoritmo realiza uma amostragem nos mapas de imagem, seguindo um raio com mesma direção e comprimento da via (linha 6), a fim de identificar se a posição da via é legal (Figura 3.8a). Caso não seja, o algoritmo realiza uma nova amostragem usando um raio ligeiramente desviado (linha 9). Ao encontrar uma posição legal, a via candidata é devidamente desviada para a nova direção (Figura 3.8b) e o algoritmo para (linhas 13 e 14).

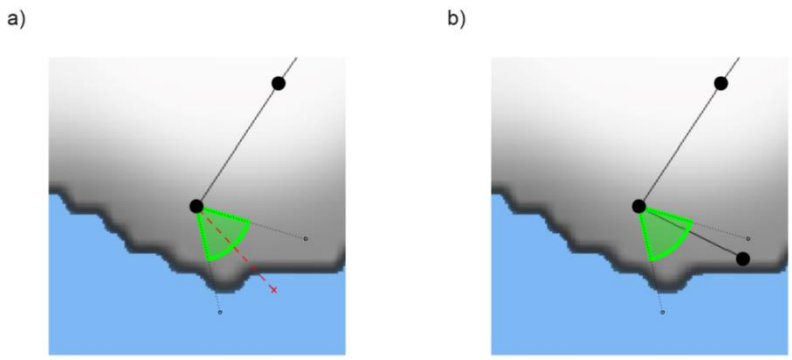

Figura 3.8 Ilustração do algoritmo de desvio de obstáculos: (a) consulta, em arco, ao mapa de corpos d'água e (b) ajuste da via para uma posição legal

Não encontrando uma posição legal com o desvio, o procedimento se repete até que uma posição legal seja encontrada ou o desvio máximo permitido seja alcançado (linha 5). Nesse último caso, o algoritmo inicia uma nova sequência de amostragens, só que com um raio ligeiramente menor (linha 11). Mais uma vez, o procedimento é repetido até que uma posição legal seja encontrada ou o tamanho mínimo para amostragem seja alcançado (linha 3).

Na avaliação de uma ramificação aplica-se o retardo, em número de derivações, que uma ramificação deve sofrer antes de se tornar uma via candidata. Esse retardo é parametrizável e pode ser usado para permitir que uma via principal se afaste suficientemente de suas ramificações antes que elas iniciem sua expansão.

Na instanciação da via principal adiciona-se, efetivamente, a via candidata à malha. Isso implica em adicionar uma nova aresta ao grafo que representa a malha. A adição de novas arestas ao grafo cuida para que não haja o cruzamento com arestas pré-existentes, do contrário o grafo se tornaria impróprio para a execução do algoritmo de extração de células rodoviárias, descrito na Seção 3.3. Para computar o cruzamento entre a nova aresta e outra, testa-se a colisão, par a par, do segmento que representa a nova aresta e o segmento que representa uma aresta pré-existente. Caso interceptem-se (Figura 3.9a), as arestas colisoras são divididas. A divisão das arestas consiste em três etapas: a criação de um vértice no ponto de interceptação (vértice divisor) (Figura 3.9b), o ajuste do vértice destino das arestas colisoras para o vértice divisor (Figura 3.9c), e a criação de novas arestas partindo do vértice divisor para os antigos destinos das arestas (Figura 3.9d).

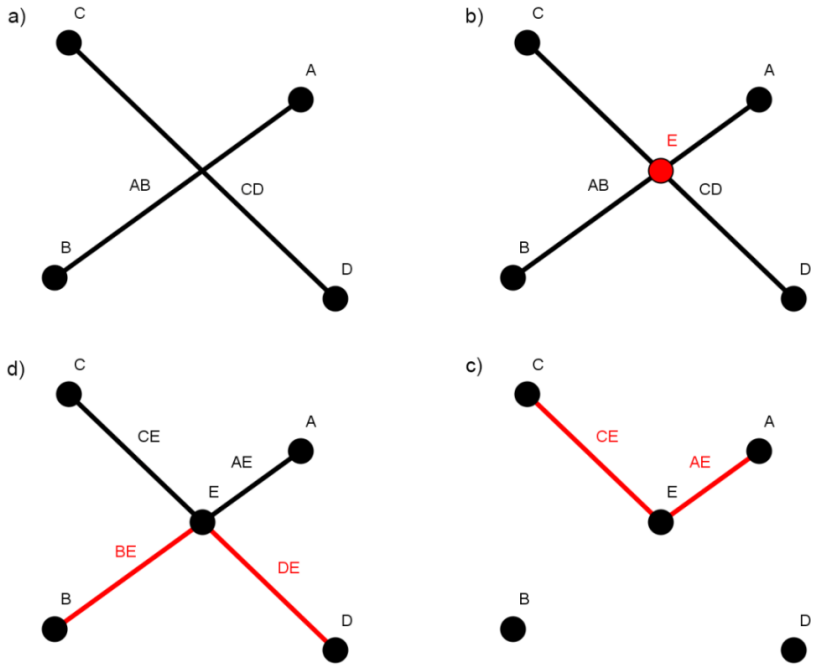

Figura 3.9 Divisão de arestas: (a) detecção da colisão, (b) criação do vértice no ponto de interseção, (c) ajuste do destino e (d) criação de novas arestas

A instanciação também pode dar continuidade à via instanciada, criando até três novos itens de trabalho: duas ramificações e uma nova via candidata. Caso se criem novos itens de trabalho, estes herdarão os atributos da via instanciada (Tabela 3.2), repassando-os para seus descendentes e, assim por diante, como em um algoritmo evolutivo.

Tabela 3.1 Atributos da via principal candidata

Atributo	Descrição	Tipo/Unidade
branchingDistance	Número de vezes que a via foi expandida	Inteiro
goal	Posição almejada pela via	Ponto no R^2
branchDepth	Número de elementos existentes desde a origem da ramificação até a via	Inteiro

Algorithm 2 Instancia Vias Principais Candidatas

```
1:  function INSTANTIATE(road, workQueues)
2:      newDepth ← road.branchDepth + 1
3:      if newDepth > max_hightway_branch_depth then
4:          return
5:      end if
6:      if road.branchingDistance == highway_branching_distance then
7:          branch1 ← NEWBRANCH(road, road.angle + 90)
8:          branch2 ← NEWBRANCH(road, road.angle - 90)
9:          ENQUEUE(workQueues[BRANCH], branch1)
10:         ENQUEUE(workQueues[BRANCH], branch2)
11:     end if
12:     if DISTANCE(road.position, road.goal) < goal_distance_threshold then
13:         road.goal ← FINDGOAL(road)
14:     end if
15:     pattern ← FINDPATTERN(road)
16:     APPLYPATTERN(pattern, road)
17:     ENQUEUE(workQueues[ROAD], road)
18: end function
```

Figura 3.10 Pseudocódigo do procedimento de instanciação de vias candidatas

O algoritmo de instanciação (Figura 3.10) começa incrementando o contador do número de vezes que a via principal foi expandida (linha 2). Caso se atinja o número máximo de expansões permitido a uma via principal (*max_hightway_branch_depth*), não se cria nenhum novo item de trabalho e a instanciação termina (linhas 3 a 5).

Em seguida, incrementa-se o contador de próxima ramificação. Caso seu valor se iguale ao parâmetro de distância de ramificação (*highway_branching_distance*), duas ramificações são criadas e o contador é reiniciado. As ramificações criadas são direcionadas pela inflexão em 90º, no sentido horário e anti-horário, da direção da via instanciada (linhas 6 a 11).

Por fim, caso a distância entre o fim da via instanciada e sua posição almejada seja maior que o parâmetro de tolerância de distância (*goal_distance_threshold*), cria-se uma nova via candidata com a mesma posição almejada. Caso não seja, considera-se que a via instanciada alcançou seu antigo objetivo e faz-se necessário buscar um novo objetivo para a via em criação (linhas 12 a 14).

A primeira parte da busca pelo objetivo de uma via é determinar onde ela deve chegar. Para encontrar essa posição, executa-se um algoritmo que localiza

a maior concentração populacional dentro das proximidades da nova via (linha 13). Esse algoritmo realiza uma amostragem de raios, a partir do final da via, em um arco de amplitude e comprimento parametrizáveis (Figura 3.11).

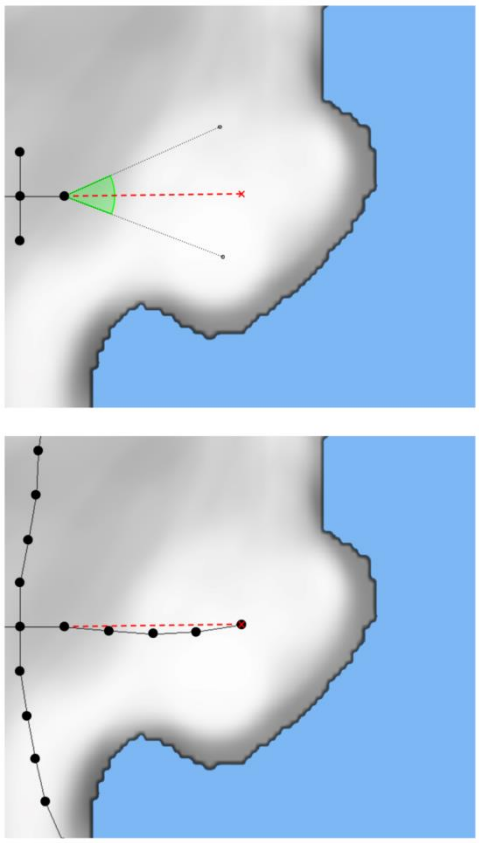

Figura 3.11 Descobrindo *onde* está o objetivo da via através da amostragem no mapa de densidades populacionais

Para cada unidade do raio, a densidade demográfica é consultada no mapa de controle. Ao percorrer todo o trajeto do raio, armazena-se a maior densidade populacional encontrada e sua respectiva posição. Os resultados são, então, computados como o produto entre a densidade populacional encontrada e sua distância para o ponto final da via. Aquele com maior peso é escolhido.

A segunda parte da busca pelo objetivo de uma via é determinar como ela chegar à posição almejada. Isso é determinado pelas características do padrão urbanístico da região sob a via. Para encontrar o padrão urbanístico sob uma via, realiza-se uma amostragem nos mapas de padrões na posição inicial da via (linha 16). Cada padrão possui características próprias que atuam na direção e

no comprimento da via. Por exemplo, o padrão de *rasterização* determina que a via se expanda apenas em angulações retas (Figura 3.12)

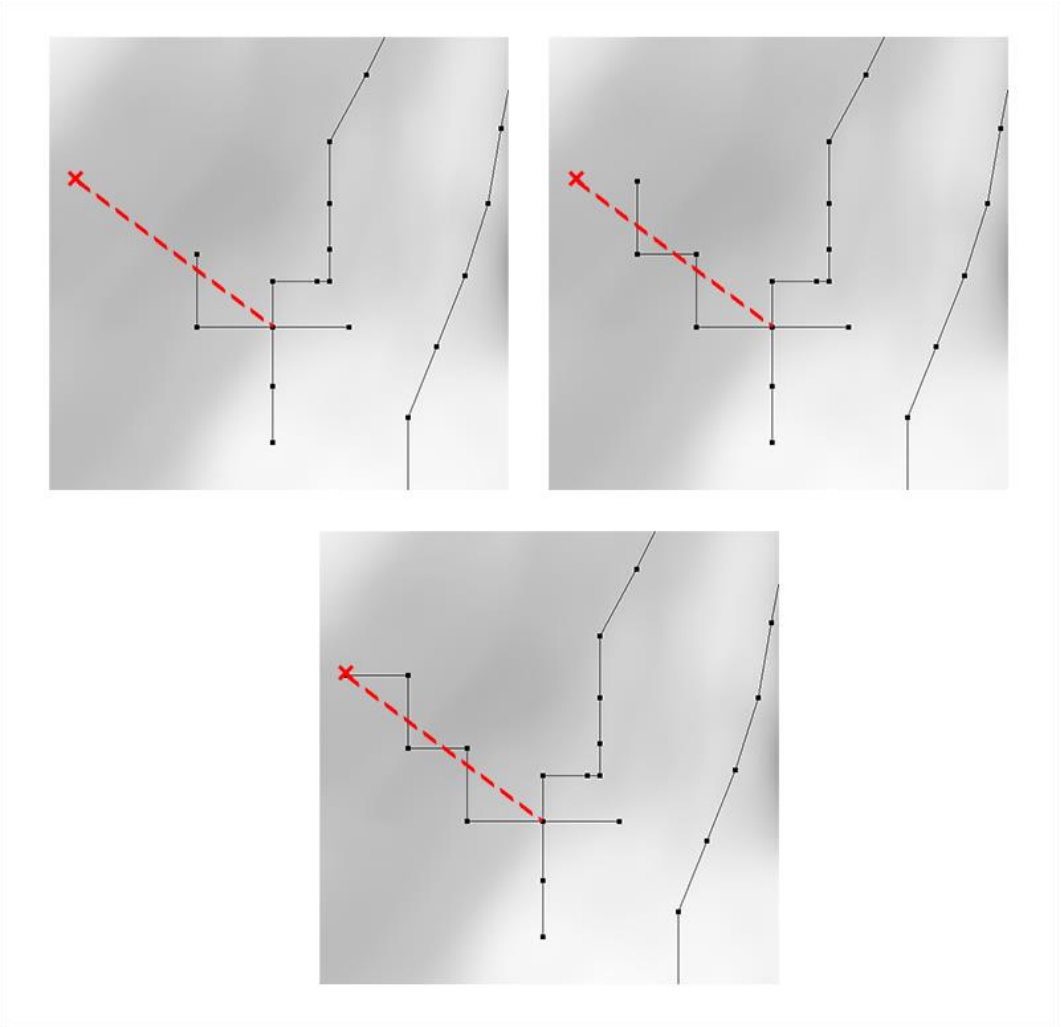

Figura 3.12 Descobrindo *como* chegar ao objetivo da via através da amostragem no mapa de padrões

Note-se que este trabalho não compreende a mistura de múltiplos padrões urbanísticos. Fuesser [10] descreve que muitas cidades reais apresentam mais do que um único padrão devido às suas diferentes fases de desenvolvimento através da história. Não sendo possível representar essa mistura de padrões encontrada em malhas rodoviárias reais, limita-se o poder de representação das malhas rodoviárias geradas.

3.3.
Extração das Células Rodoviárias

Após a expansão da malha rodoviária primária é possível computar os nichos, ou células rodoviárias, onde a expansão da malha rodoviária secundária ocorrerá. A extração das células rodoviárias é realizada através da identificação dos ciclos de base mínima (*minimum cycle basis*) do grafo que representa a malha.

Essa identificação é realizada através do algoritmo apresentado em [7]. Como será mostrado, esse algoritmo se assemelha muito ao de árvore de extensão mínima (*minimum spanning tree*), podendo ser entendido como uma derivação dele.

O algoritmo inicia pela ordenação de todos os vértices do grafo. Os vértices são dispostos em um vetor de acordo com suas posições no eixo das abscissas, do menor valor para o maior, e como critério de desempate, de acordo suas posições no eixo das ordenadas (também do menor para o maior). O caminhamento começa, então, visitando o primeiro vértice do vetor ordenado, que é o vértice que se encontra mais remotamente à esquerda e abaixo. O algoritmo segue visitando sua adjacência mais próxima no sentido horário. Após visitá-lo, o caminhamento prossegue pela nova adjacência mais próxima no sentido anti-horário.

Caso um vértice previamente visitado seja visitado novamente, significa que um ciclo mínimo foi encontrado. Se o caminhamento for interrompido por não haver mais adjacências a visitar, significa que um filamento foi percorrido.

Ao encontrar um ciclo o algoritmo deve extraí-lo do grafo, mas não o faz indiscriminadamente. Antes, procura identificar se alguma de suas arestas pode ser compartilhada com outro ciclo, não mínimo. Se houve alguma, ao invés de removê-la o algoritmo apenas a sinaliza para que seja removida posteriormente.

Após a execução do algoritmo, inicia-se a extração das células rodoviárias. O bordo do polígono que representa o ciclo é usado como limite de expansão para as vias secundárias (Figura 3.13a). Uma caixa envolvente orientada (*OBB*) é computada a partir do bordo convexo formado pelos vértices do ciclo (Figura 3.13b). O maior eixo da *OBB* e o centroide do ciclo são usados para determinar a orientação e a posição iniciais das vias secundárias que expandirão dentro dessa célula (Figura 3.13c e d).

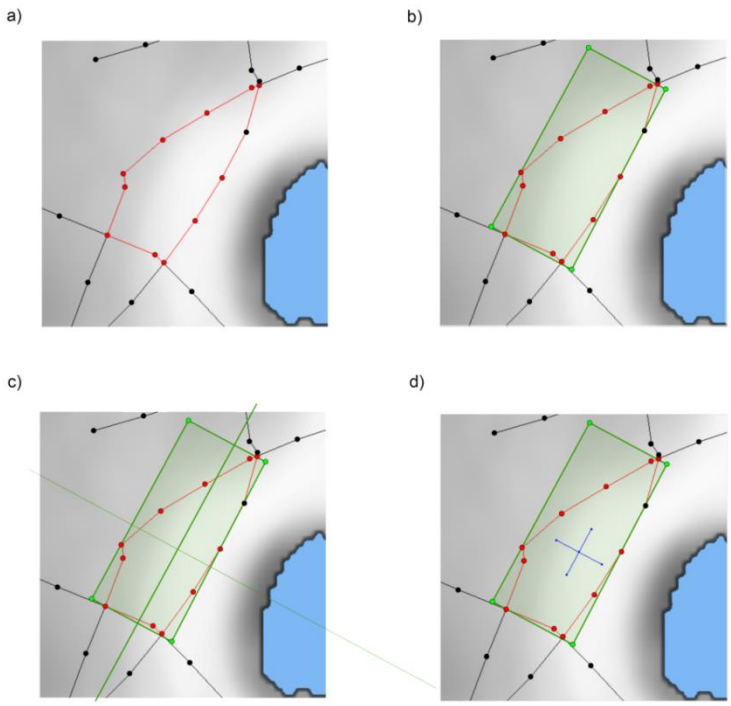

Figura 3.13 Extração da célula rodoviária: (a) identificação do bordo do ciclo, (b) computação de uma OBB, (c) orientação das vias secundária e (d) posicionamento das vias secundárias no centroide do ciclo

3.4. Expansão da Malha Rodoviária Secundária

A expansão da malha rodoviária secundária é bastante similar à expansão da malha primária. Ela começa com a criação de quatro vias locais candidatas para cada célula rodoviária extraída. Essas vias são avaliadas e instanciadas, gerando novas vias candidatas.

A expansão da malha secundária é composta por apenas dois procedimentos: a avaliação das vias locais e a instanciação das vias locais. Assim como na expansão da malha primária, os procedimentos também são invocados em derivações. A expansão da malha secundária realiza tantas derivações quantas definidas pelo parâmetro *num_street_derivations*.

A avaliação das vias locais é muito similar à avaliação das vias principais, diferindo destas apenas por criar vias candidatas locais. Já a instanciação das vias locais difere significativamente da sua contraparte da expansão da malha primária. Ao adicionar a via local à malha, e consequentemente ao grafo, cuida-

se para que a sua aresta não cruze as arestas das vias primárias que compõem o bordo da célula rodoviária que a cerca (Figura 3.14).

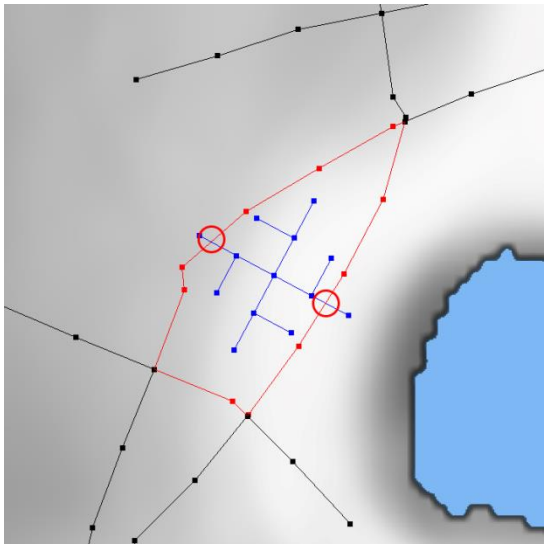

Figura 3.14 Vias locais não devem cruzar as arestas do bordo da célula rodoviária

Para computar o cruzamento entre a nova aresta e as arestas do bordo, testa-se a colisão, par a par, entre o segmento de reta que representa a nova aresta e o segmento de reta que representa uma aresta do bordo. Caso interceptem-se, a aresta do bordo é dividida e a aresta da via local é interrompida no ponto de interceptação (Figura 3.15).

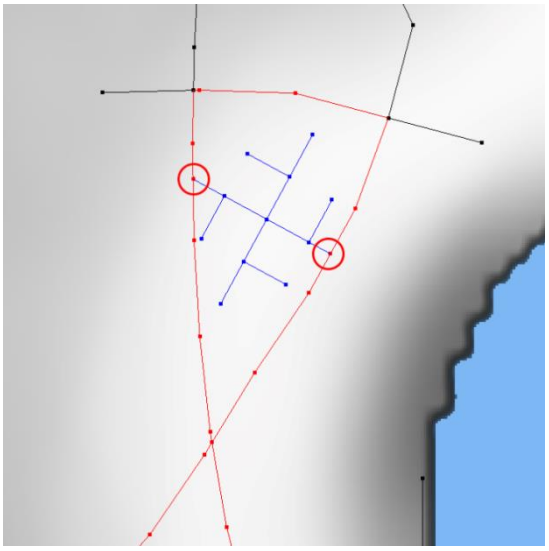

Figura 3.15 A aresta do bordo é dividida pela via local em expansão

Caso a aresta da via local seja interrompida, a instanciação termina. Do contrário, ela pode dar continuidade à via instanciada criando até três novas vias locais candidatas. Novamente, os atributos da via instanciada (Tabela 3.3) são herdados pelas novas vias a serem criadas.

Tabela 3.2 Atributos da via local candidata

Atributo	Descrição	Tipo/Unidade
branchDepth	Número de vezes que a via foi expandida	Inteiro
boundsIndex	Índice do bordo da célula rodoviária que limita a expansão da via local	Inteiro
pattern	Código do padrão urbanístico da via local	Inteiro
childCode	Código que corresponde a uma das três possibilidades de expansão da via local	Inteiro/[0..2]

A instanciação continua incrementando o contador do número de vezes que a via foi expandida. Caso esse contador se iguale ao número máximo de vezes que uma via local pode ser expandida (*max_street_branch_depth*), não se cria nenhuma nova via candidata e a instanciação termina.

Caso a instanciação não tenha sido terminada, faz-se necessário definir o *objetivo* das novas vias. No caso das vias locais, o objetivo é ditado unicamente pelas características do padrão urbanístico da região. Para encontrar o padrão urbanístico da região, realiza-se uma amostragem nos mapas de padrões na posição inicial da via. Cada padrão determina as direções e comprimentos das vias candidatas criadas. Nesse trabalho, implementou-se apenas o padrão *checkerboard*, que determina a criação de vizinhanças quadradas, como as marcações de um tabuleiro de damas (Figura 3.16).

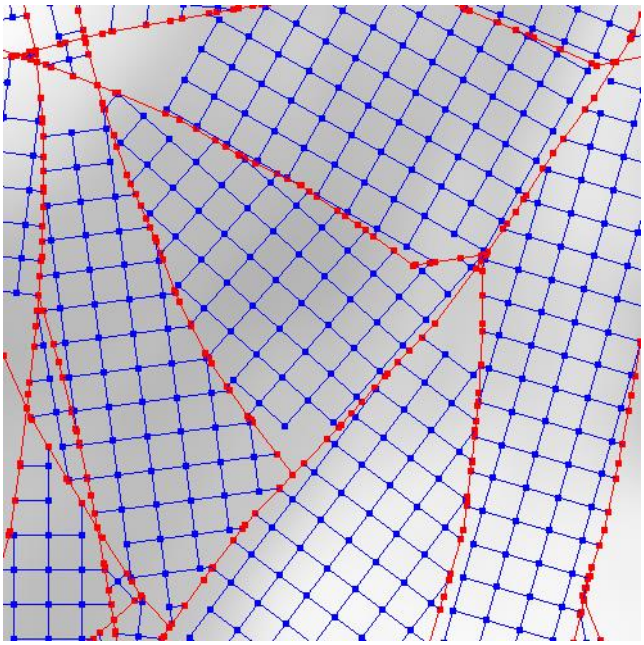

Figura 3.16 Vizinhanças quadradas geradas segundo o padrão *checkerboard*

Note-se que não há teste de colisão entre arestas de vias locais. Isso ocorre, pois, sabendo o padrão e qual das três vias está sendo instanciada, é possível determinar como conectar uma via a outras sem precisar consultar todo o grafo. No padrão implementado, as vias se expandem para cima e para a esquerda. A via que expande para a direita de seu pai (Figura 3.17a) deve conectar-se (Figura 3.17b) à via que expandira a esquerda (Figura 3.17c) daquela que expandiu segundo seu avô.

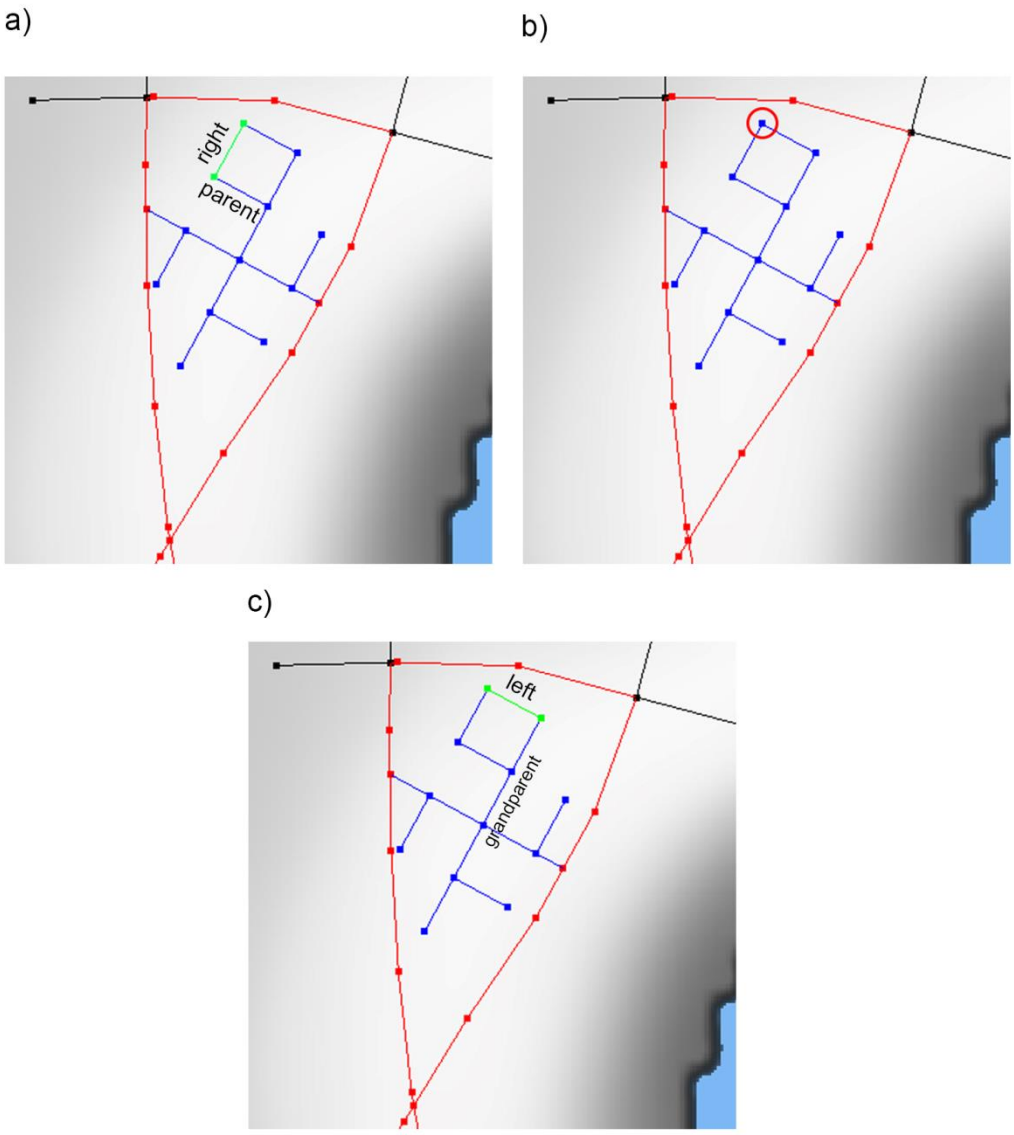

Figura 3.17 (a) A via que se expande para a direita de seu pai, (b) conecta-se à (c) via que se expande a esquerda daquela se expandiu segundo seu avô.

3.5. Implementação na GPU

Para que se pudesse apurar as vantagens da execução em dispositivos massivamente paralelos, este trabalho implementou o algoritmo proposto em duas plataformas distintas. A maior parte do algoritmo foi implementando tanto na GPU (arquitetura *throughput-oriented*) quanto na CPU (arquitetura *latency-oriented*). Considerou-se fora do escopo deste trabalho implementação do

algoritmo de extração das células na GPU e, por isso, essa funcionalidade fora implementada apenas na CPU.

A arquitetura *throughput-oriented* das GPUs enfatiza a execução lenta de muitos processos concorrentes, ao contrário da execução rápida de poucos processos, típica das arquiteturas *latency-oriented* [13]. Por causa disso, partes da implementação na GPU diferem significativamente da implementação na CPU. Essa diferenciação ocasionou a adição de alguns detalhes ao processo de geração, conforme mostrado na Figura 3.18.

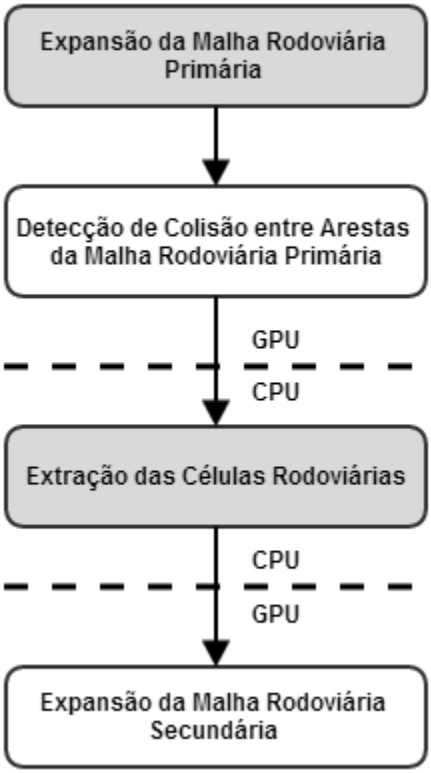

Figura 3.18 Fluxograma das partes do processo de geração na GPU

Crê-se, ainda, que os detalhes dessas implementações diferenciadas estejam intimamente relacionados aos resultados obtidos no Capítulo 4 e, por essa razão, eles serão descritos nas Seções a seguir.

Para implementar o algoritmo na GPU, usou-se CUDA. CUDA é uma plataforma de programação paralela implementada pelas GPUs NVIDIA. As GPUs NVIDIA possuem três arquiteturas de hardware diferentes. Da mais moderna para a mais antiga, elas são: Kepler, Fermi e Tesla [23]. Ao desenvolver a implementação na GPU, visou-se fazê-lo na arquitetura Kepler.

Nada impede, no entanto, que o algoritmo seja implementado nas arquiteturas mais antigas, obtendo-se resultados com desempenho ligeiramente inferior.

3.5.1. *Kernel* de Expansão da Malha

O *kernel* de expansão foi implementado segundo a estratégia de *threads* persistentes [12]. Para isso, o *kernel* implementa um algoritmo *lock-free* que coordena a expansão das malhas rodoviárias iterativamente sem que seja necessário invocá-lo múltiplas vezes da CPU. O algoritmo usado é o mesmo para ambas as malhas, mudando-se apenas as filas de trabalho em cada caso. Como mencionado anteriormente, usa-se duas listas de filas de trabalho de maneira alternada, de forma similar à técnica de *double buffering*.

A execução em grade é organizada da seguinte maneira: cada bloco B_M escolhe uma fila de trabalho, da lista frontal de N filas, e cada *thread* por um item I_k da fila escolhida, onde$K = [M \ mod \ N]$.

A estrutura básica do *kernel* de expansão são dois laços aninhados (Figura 3.19). O laço externo itera por todas as derivações (linhas 8 a 37), enquanto o laço interno (linhas 15 a 29) realiza o processamento dos itens de trabalho até que não haja mais itens na fila escolhida.

Algorithm 6 Kernel de Expansão

```
Require: counter == 0                                          ▷ Global Memory
 1: function EXPAND(front, back, numDerivations, start, end)
 2:     if threadIdx.x == 0 then
 3:         state ← 0                                          ▷ Shared Memory
 4:         derivation ← 0                                     ▷ Shared Memory
 5:         i ← start + (blockIdx.x%end)                       ▷ Shared Memory
 6:     end if
 7:     SYNCTHREADS()
 8:     while derivation < numDerivations do
 9:         if state == 0 then
10:             if threadIdx.x == 0 then
11:                 state ← 1
12:                 ATOMICADD(&counter, 1)
13:             end if
14:             SYNCTHREADS()
15:             while state == 1 do
16:                 if threadIdx.x == 0 then
17:                     RESERVEDEQUEUE(front[i], blockDim.x, head, reserved)
18:                 end if
19:                 SYNCTHREADS()
20:                 if threadIdx.x < reserved then
21:                     j ← head + threadIdx.x
22:                     PROCESSWORKITEM(front, i, j, back)
23:                 end if
24:                 if threadIdx.x == 0&&reserved == 0 then
25:                     state ← 2
26:                     ATOMICSUB(&counter, 1)
27:                 end if
28:                 SYNCTHREADS()
29:             end while
30:         end if
31:         if threadIdx.x == 0&&counter == 0 then
32:             state ← 0
33:             derivation + +
34:             SWAP(front, back)
35:         end if
36:         SYNCTHREADS()
37:     end while
38: end function
```

Figura 3.19 Pseudocódigo do *kernel* de expansão

Para garantir que todas as *threads* processem itens pertencentes à mesma derivação, usa-se um conjunto de variáveis. Essas variáveis são: o valor da derivação corrente e do estado desta derivação - armazenadas em memória compartilhada - e um contador - armazenado em memória global. As variáveis armazenadas em memória compartilhada são manipuladas apenas pela primeira

thread de cada bloco. Garante-se que essas variáveis sejam lidas de maneira correta pelas outras *threads* do mesmo bloco através do uso de barreiras de memória (linhas 7, 19, 28 e 36).

O algoritmo começa configurando o valor da derivação corrente e o estado da derivação com seus valores iniciais (linhas 3 e 4). O laço externo inicia caso a derivação corrente seja menor do que a derivação máxima a ser alcançada. Verifica-se o estado da derivação e, caso seja igual ao estado inicial, incrementa-se o contador atomicamente, uma vez para cada bloco, e atribui-se um segundo valor para o estado (linhas 10 a 13). O laço interno inicia, realiza-se a retirada coordenada de itens da fila de trabalho (linhas 16 a 18) e continua-se a iterar por ele até que a fila esteja vazia (linha 24 a 27). Ao sair do laço interno, verifica-se o contador e, se este for igual a zero - significando não há mais blocos processando itens de trabalho - incrementa-se a derivação corrente, alterna-se as listas de filas de trabalho e retorna-se ao estado inicial (linhas 31 a 35). A verificação na linha 9 garante que, enquanto não se retornar ao estado inicial, o processamento dos itens de trabalho é executado.

3.5.2. Filas de Trabalho

Como discutido anteriormente, o uso de filas de trabalho pretende facilitar o tratamento de questões que impactam na eficiência de um algoritmo paralelo: sincronismo, heterogeneidade e dinamismo (Seção 3.1.3).

Sincronismo refere-se ao mecanismo de segurança empregado para garantir a segurança no acesso aos dados compartilhados entre *threads*. Apesar de ser uma questão que extrapola o uso das filas de trabalho, as filas de trabalho devem permitir, ao menos, a adição e remoção segura de elementos em paralelo. Neste trabalho implementou-se o enfileiramento e o desenfileiramento seguros *non-blocking*. Por usar operadores atômicos e barreiras de memória durante a adição de itens, não se garante o progresso ininterrupto das *threads* e, como resultado, essa implementação é caracterizada apenas como livre de exclusão mútua (*lock-free*).

Algorithm 4 Enfileira e Desenfileira Itens na Fila de Trabalho

```
1: function ENQUEUE(queue, item)
2:     if count >= max_num_workitems then
3:         return false
4:     end if
5:     mask ← BALLOT(1)
6:     activeThreads ← POPC(mask)
7:     currThread ← LANEMASK_LT()&mask
8:     laneId ← POPC(currThread)
9:     leadingThread ← FFS(mask) − 1
10:    if laneId == leadingThread then
11:        ATOMICADD(&count, activeThreads)
12:        first ← ATOMICADD(&tail, activeThreads)
13:    end if
14:    first ← SHFL(first, leadingThread)
15:    index ← ((first + laneId)%max_num_workitems)
16:    queue.data[index] ← item
17:    THREADFENCE()
18:    return true
19: end function
20: function DEQUEUE(queue, index)
21:            ▷ Dequeue is wait-free because of ReserveDequeue (See Ahead)
22:    return queue.data[index%max_num_workitems]
23: end function
```

Figura 3.20 Pseudocódigo do enfileiramento e desenfileiramento de itens na fila de trabalho

O algoritmo de enfileiramento (Figura 3.20) emprega uma otimização no nível do *warp*[1] a fim de diminuir a latência no uso dos operadores atômicos. Ele começa identificando qual é o número da raia, dentro do *warp*, ocupada pela *thread* (linhas 5 a 9). Apena a *thread* que encabeça as raias incrementa atomicamente o contador de itens e a cauda da fila no número de *threads* ativas (linhas 10 a 13). Ao incrementar a cauda da fila, obtém-se a posição inicial para a inserção no vetor de itens. A partir daí, todas as *threads* do *warp* calculam a posição final de inserção do item, deslocando a posição inicial pelo número da raia da *thread* (linha 14 e 15). Após a inserção do item na fila, usa-se uma barreira de memória para garantir que a execução do *warp* só prosseguirá quando o item for efetivamente escrito na memória global (linha 17).

A Figura 3.20 mostra, também, o uso da função intrínseca *__shfl()*, disponível apenas na arquitetura Kepler. A intrínseca *__shfl()* permite para

[1] Em CUDA um *warp* corresponde a um grupo de *threads* executando concorrentemente em um mesmo multiprocessador.

passar à todas as *threads* de um mesmo *warp* um valor armazenado nos registradores de uma delas. Como uma alternativa ao uso da __*shfl()*, sugere-se usar memória compartilhada com barreiras de memória.

Nas GPUs, a heterogeneidade está relacionada à arquitetura SIMT (única instrução, múltiplas *threads*). Na arquitetura SIMT existe perda de desempenho no caso de divergências de fluxo de controle. Entende-se por divergência de fluxo de controle a execução de partes distintas de código por *threads* de um mesmo *warp*. Como se executa uma única instrução por *warp*, executar partes distintas de código (instruções distintas) implica no desativamento temporário de algumas *threads* (Figura 3.21).

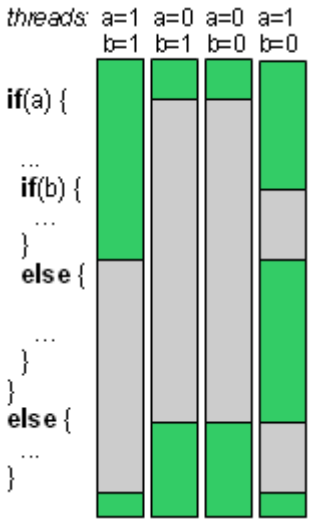

Figura 3.21 A execução de instruções distintas por uma *thread* leva ao seu desativamento temporário

É desejável, então, que o maior número possível de *threads* execute o mesmo código. No contexto das filas de trabalho esse desejo se traduz como o desenfileiramento concorrente de itens relacionados ao mesmo código. Para fazê-lo, implementou-se uma estratégia de desenfileiramento coordenado de itens de trabalho. Resumidamente, essa estratégia determina que a primeira *thread* de cada bloco[2] reserve um número de itens de trabalho igual ao tamanho do bloco, disponibilizando-os para as outras *threads*.

O algoritmo de reserva de desenfileiramentos (Figura 3.22) inicia propriamente com o decremento atômico do contador de número de itens da fila

[2] Da perspectiva do programador, a API CUDA organiza a computação paralela hierarquicamente em *threads*, blocos e grades. Uma *thread* corresponde a um processo paralelo, um bloco a um grupo de *threads* e a grade a um grupo de blocos.

(linha 6). Caso o contador desça abaixo de zero (ou seja, caso haja mais *threads* do que itens disponíveis para retirada) restitui-se o contador pela diferença de itens não reservados (linhas 7 a 11). Em seguida, comunica-se à cada *thread* a quantidade de itens reservados (linhas 12 a 16). Comparando seu identificador único com o número de reservas feitas, cada *thread* é capaz de saber se tem ou não um item reservado para ela e, assim, retirá-lo sem necessidade de nenhum cuidado especial.

Algorithm 5 Reserva Desenfileiramentos na Fila de Trabalho

```
1:  function ReserveDequeue(queue, reserves, &reserved, &first)
2:      if queue.count <= 0 then
3:          reserved ← 0
4:          return false
5:      end if
6:      oldCount ← atomicSub(&queue.count, reserves)
7:      if oldCount > 0 then
8:          overflow ← min(oldCount - reserves, 0)
9:          if overflow < 0 then
10:             atomicSub(&queue.count, overflow)
11:         end if
12:         reserved ← reserves + overflow
13:         first ← atomicAdd(&queue.head, reserved)
14:     else
15:         numReserves ← 0
16:         atomicAdd(&queue.count, reserves)
17:     end if
18: end function
```

Figura 3.22 Pseudocódigo da reserva de desenfileiramentos na fila de trabalho

Apesar da criação dinâmica de novos itens de trabalho durante a execução, a implementação de fila deste trabalho usa memória estática. Usa-se a memória pré-alocada na forma de buffers circulares. A principal vantagem dos buffers circulares é que não é necessário reorganizá-los caso seus elementos sejam removidos. Comparadas às alternativas dinâmicas, as filas estáticas circulares são bastante eficientes [30]. No entanto, essa decisão impõe a pré-alocação da memória em quantidade suficiente para que se comportem todos os itens de trabalho, inclusive os adicionados dinamicamente durante a execução.

3.5.3.
Amostragem de Raios

Durante os procedimentos e avaliação e instanciação da via candidata, na expansão da malha rodoviária primária, realizam-se amostragens em raio aos mapas de imagem (Seção 3.2). Otimizou-se essa amostragem de forma que se pudessem antecipar vários acessos à memória global em um único acesso (*memory coalescing*). As amostragens são realizadas, uma por *thread*, caminhando-se sobre um segmento de reta. Como os acessos concorrentes à memória obedecem a uma coerência espacial, a estratégia de cache 2d das memórias de textura permite antecipar múltiplos acessos à memória em apenas uma transação (Figura 3.23).

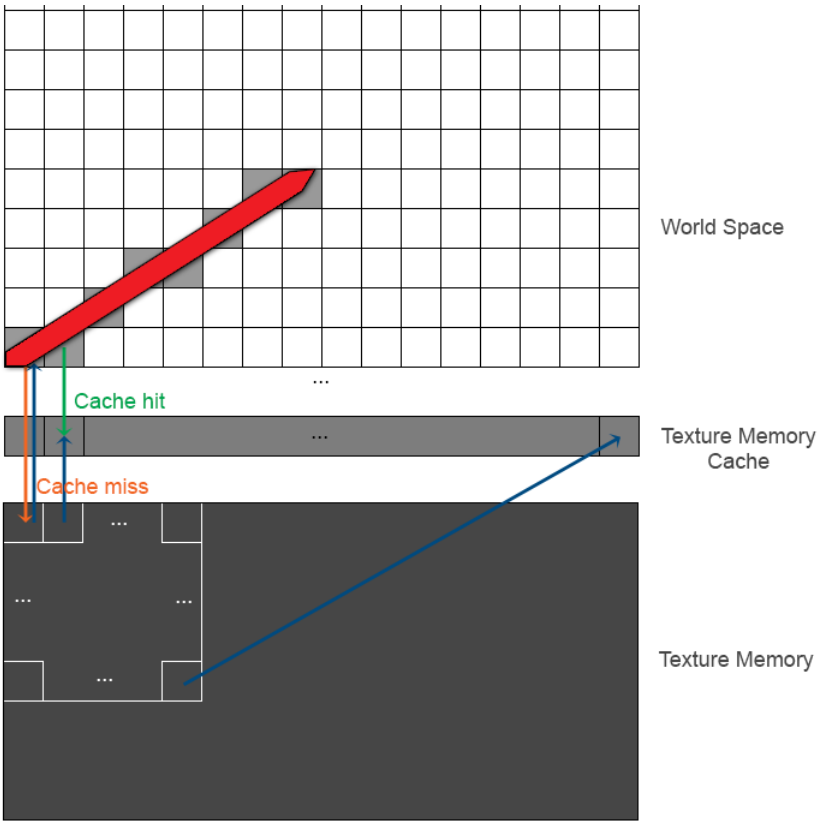

Figura 3.23 A estratégia de cache da memória de textura permite acessar múltiplos endereços de memória em uma única transação

Como na conversão das coordenadas de espaço de mundo para coordenadas de textura perde-se precisão, é interessante a aplicação de uma

estratégia de *anti-aliasing* nas amostragens. Felizmente, a memória de textura provê a aplicação de filtros (ex.: filtro linear) de baixo custo, acelerada por hardware [9].

Para usar a memória de textura, copiam-se os mapas de imagens para objetos de textura. A fim de reduzir o tamanho dos objetos de textura, usou-se o formato de dados *unsigned char* com apenas um canal de cor. Para compatibilizar os mapas com esse formato, os dados de cor são convertidos para níveis de cinza e normalizados entre 0 e 255.

Juntamente com o uso da memória de textura, usa-se um cache local em memória compartilhada. Como o algoritmo de amostragem de raios realiza uma amostra para cada unidade do raio, o número de amostras, e consequentemente o número de itens em cache, é igual a diferença do comprimento máximo pelo comprimento mínimo do arco (*max_sampling_ray_length* - *min_sampling_ray_length*).

Para garantir que esse cache seja realmente alocado em memória compartilhada, o limite da diferença entre os comprimentos é estipulado em 1024 unidades.Com esse limite, o cache ocupa, no máximo,2Kbytes (1024 * 2 bytes por amostragem), muito abaixo do piso de memória compartilhada dos dispositivos modernos[3].

Tão logo os valores são consultados em memória de textura, eles são armazenados em memória compartilhada para que sejam acessados mais rapidamente em operações subsequentes.

3.5.4. *Kernel* de Detecção de Colisão

Na expansão das malhas, primária e secundária, faz-se necessário detectar as colisões entre as vias. Contudo, na implementação na GPU, a detecção de colisões das vias principais (malha rodoviária primária) foi movida para uma etapa posterior. Isso foi feito a fim de se estabelecer um padrão de acesso à memória global favorável ao acesso concorrente.

Note que a interceptação das vias, ou a detecção de colisão entre os segmentos de reta que as representam, não ocasiona a parada de expansão, pois, do contrário, o algoritmo se tornaria dependente de ordem.

[3] A partir da *compute capability* 2.0, os dispositivos gráficos NVIDIA dispõem, no mínimo, de 48 Kbytes de memória compartilhada para cada multiprocessador [23].

O algoritmo de detecção de colisão usa a estrutura de dados *quadtree* (Figura 3.24) para minimizar o número de testes de colisão e organizar a sua execução em grade.

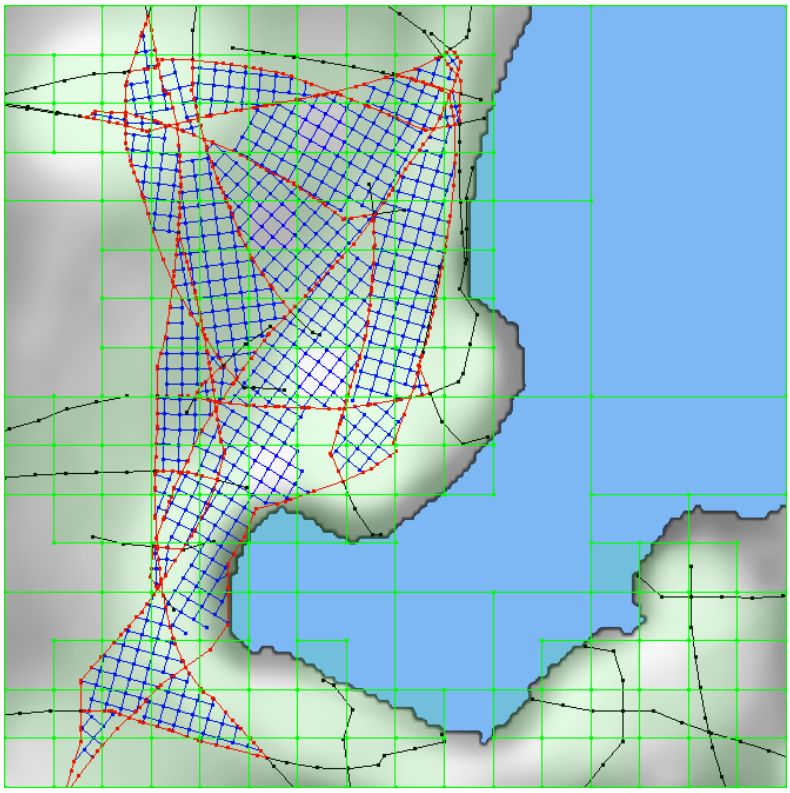

Figura 3.24 A estrutura de dados *quadtree* divide o espaço do mapa e guarda referência às arestas do grafo de acordo com suas posições

A execução em grade do algoritmo é organizada da seguinte maneira: cada bloco se responsabiliza por um nó folha da *quadtree* (quadrante) e cada *thread* T_M por um subconjunto F_M pertencente conjunto de arestas do nó folha F_N, de forma que:

$$\forall\, a \in F_M, \exists\, b_L \in F_N \mid a = b_L \; se \; L \bmod M = 0$$

Uma *thread* T_M testa a colisão entre as arestas$a_i \in F_M$e as arestas$a_j \in F_N$.Caso uma colisão seja detectada, dividem-se as arestas colisoras, conforme descrito na Seção 3.2. Para garantir que não haja alterações concorrentes durante a divisão, usa-se um esquema de *locks* duplas. Para evitar *deadlocks* por causa do uso de *locks* duplas (Figura 3.25a), uma *thread* computa apenas as colisões entre a_i e a_j onde i > j (Figura 3.25b).

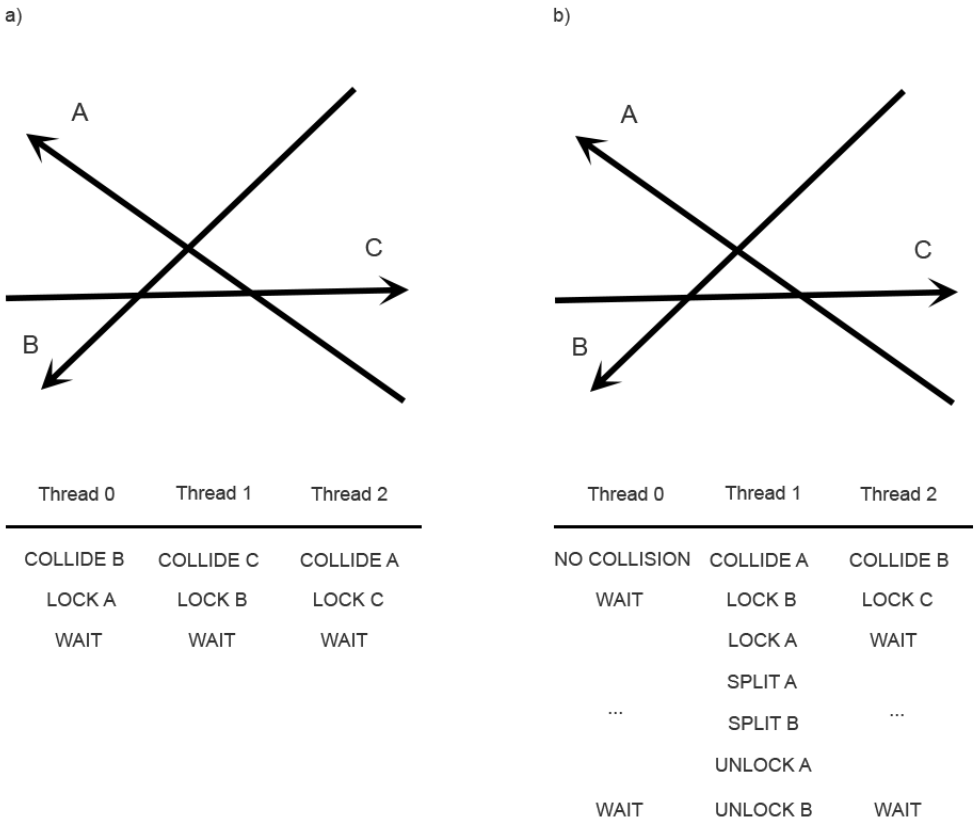

Figura 3.25 (a) Sequência de instruções com *locks* duplos suscetível ao *deadlock* / (b) Sequência de instruções com *locks* duplos e tratamento do problema de *deadlock*

Uma nova aresta a_k, adicionada a partir da divisão das arestas colisoras, também é adicionada à *quadtree*. Como a_k é inserida ao final do grupo F_M, ela é testada contra todas as arestas adicionadas antes dela.

A execução em grade se relaciona com a partição espacial estabelecida pela *quadtree*, de forma que a cada quatro blocos processem-se quatro quadrantes dispostos de maneira contígua em memória. A estratégia de cache linear da memória global permite a antecipação de múltiplos acessos à memória em um único acesso, similarmente à Seção 3.5.3. Note que, neste caso, preferiu-se usar a memória global ao invés da memória de textura não só por causa do padrão de acesso, mas por causa do tamanho dos caches disponíveis para cada tipo de memória[4]. Como os quadrantes armazenados em cache possuem, em média, 2 Kbytes (500 referências às arestas * 4 bytes), o tamanho

[4] A partir da *computer capability 2.0*, memórias de textura dispõem de, no mínimo, 8 Kbytes de cache *versus* os mínimos 48 KBytes do cache L2 [23].

do cache é determinante para a otimização do acesso à memória, pois limita o número de quadrantes que serão armazenados em uma única transação.

4
Resultados

Neste capítulo apresentar-se-á uma análise detalhada da execução do algoritmo proposto na GPU. Os critérios de análise escolhidos foram: o impacto do aumento da carga de trabalho, o impacto da granularidade da divisão espacial, o impacto do aumento do paralelismo e o impacto visual das mudanças feitas ao modelo de Parish et al. Ao fim do capítulo, compara-se a execução na GPU com a execução do mesmo algoritmo na CPU. A implementação na CPU é idêntica à implementação na GPU, salvo as particularidades descritas na Seção 3.5.

Construíram-se os gráficos com as médias ponderadas das execuções do algoritmo, todas com os mesmos conjuntos de parâmetros. O uso dessas médias visa mitigar a interferência de fatores não determinísticos na computação dos resultados, como a ordem de execução das *threads*.

Computaram-se os resultados em dois cenários. O primeiro deles retrata o entorno da enseada de Botafogo e é referenciado nos gráficos como *botafogo bay*. O segundo retrata todo o município do Rio de Janeiro e é referenciado como *rio de janeiro city*. Cada cenário possui características distintas que influenciam a execução do algoritmo. O cenário da enseada de Botafogo apresenta uma vasta área desobstruída para a expansão da malha rodoviária e é limitado por águas em apenas um dos lados. Já o cenário do Rio de Janeiro possui diversos acidentes topográficos que obstruem a expansão da malha rodoviária e é limitado por águas, ficticiamente, em todos os lados.

4.1.
Impacto do aumento da carga de trabalho

No algoritmo proposto, a carga de trabalhos e refere ao tamanho da malha a ser gerada. Como o algoritmo não provê nenhum controle direto sobre o tamanho da malha, usou-se o número de passos da simulação, mencionado na Seção 3.1.2 (parâmetros *num_highway_derivations* e *num_street_derivations*).

Apesar de indireto, esses parâmetros possibilitam o aumento determinístico e incremental da carga, o que é suficiente para efetuar a análise pretendida.

Como não é possível controlar o crescimento da geração de arestas em intervalos igualmente espaçados, não é possível observar diretamente a relação entre o número de derivações e o tempo de execução. No entanto, é possível observar a relação entre o número de derivações e o tempo de execução (Figura 4.1 a Figura 4.4).

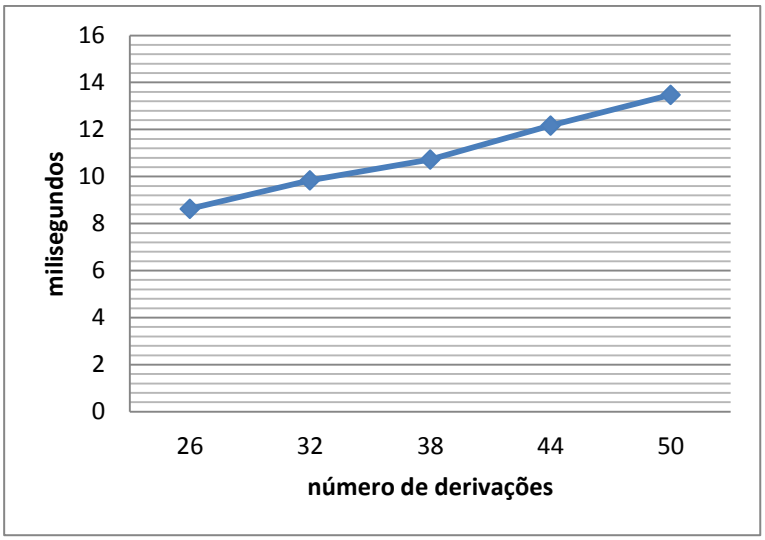

Figura 4.1 Número de derivações da malha primária x Tempo de execução (*botafogo bay*)

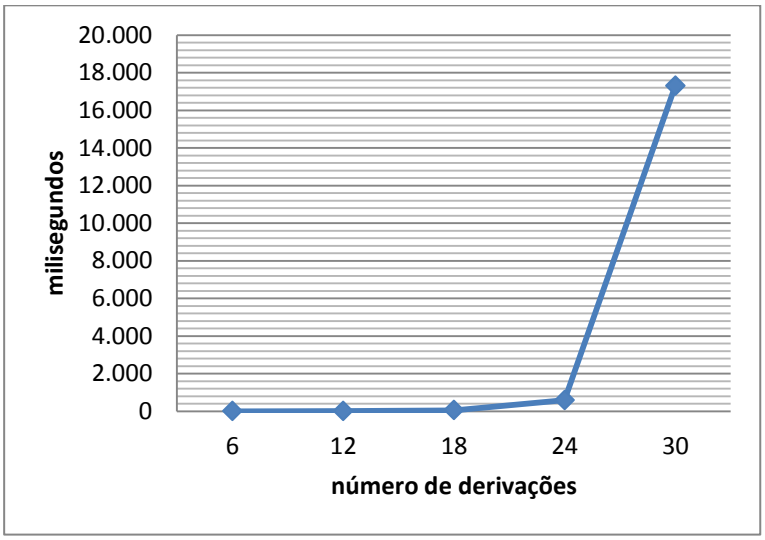

Figura 4.2 Número de derivações da malha secundária x Tempo de execução (*botafogo bay*)

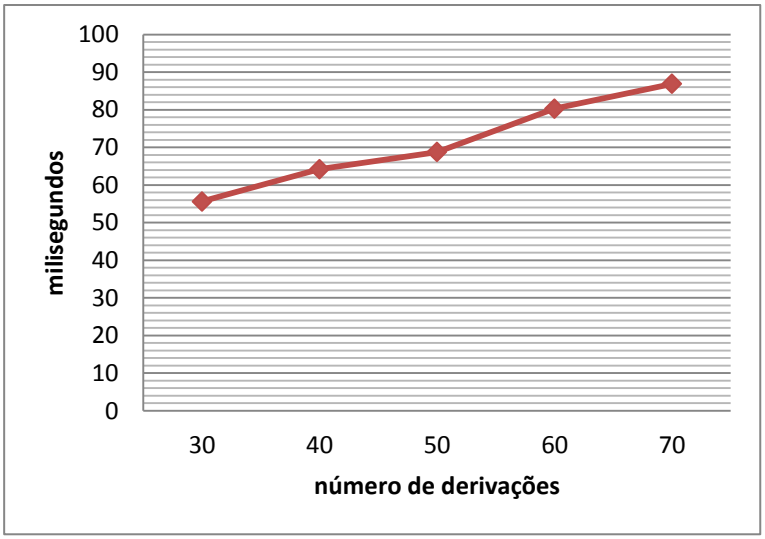

Figura 4.3 Número de derivações da malha primária x Tempo de execução (*rio de janeiro city*)

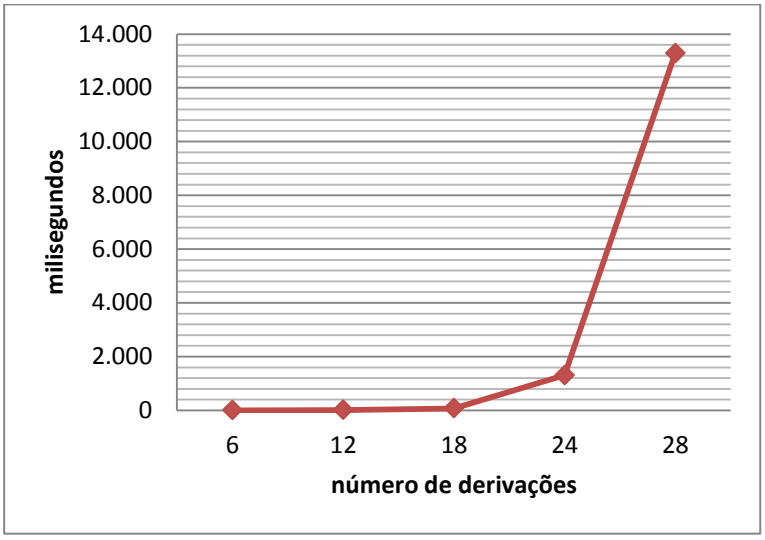

Figura 4.4 Número de derivações da malha secundária x Tempo de execução (*rio de janeiro city*)

As curvas nas Figura 4.1 e Figura 4.3 mostram que o número de arestas geradas na expansão da malha primária cresce quase que linearmente com o aumento do número de derivações. Já as curvas nas Figura 4.2 e Figura 4.4 mostram que o número de arestas geradas na expansão da malha secundária cresce exponencialmente com o aumento do número de derivações.

Em seguida, observa-se a relação entre número de arestas e tempo de execução nas Figura 4.5 a Figura 4.8.

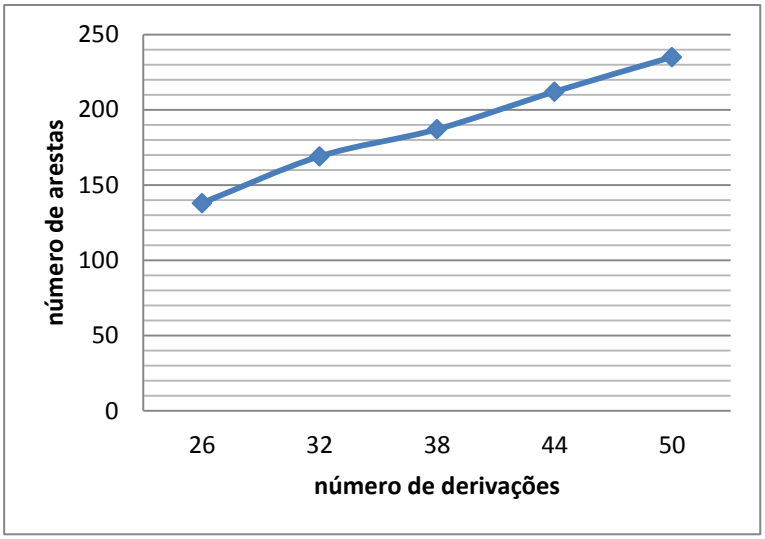

Figura 4.5 Número de derivações da malha primária x Número de arestas (*botafogo bay*)

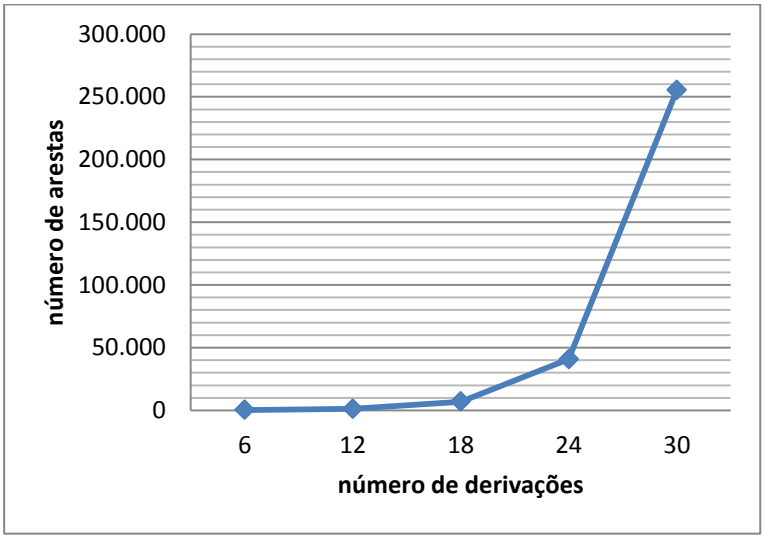

Figura 4.6 Número de derivações da malha secundária x Número de arestas (*botafogo bay*)

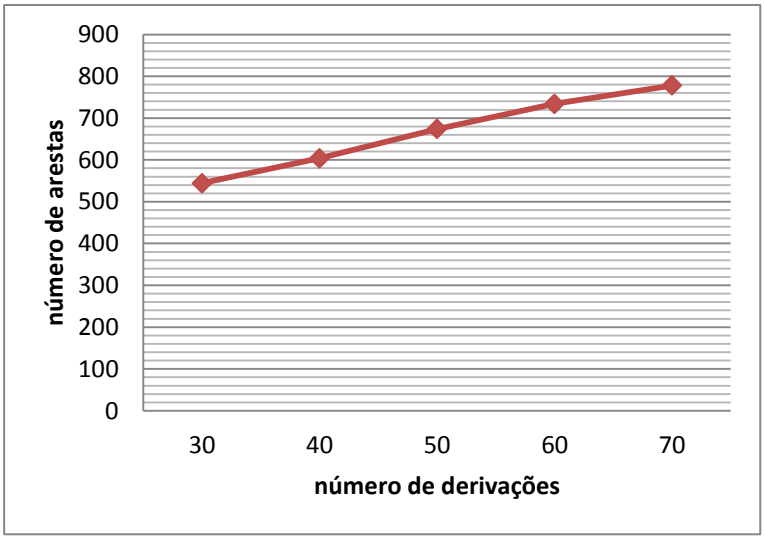

Figura 4.7 Número de derivações da malha primária x Número de arestas (*rio de janeiro city*)

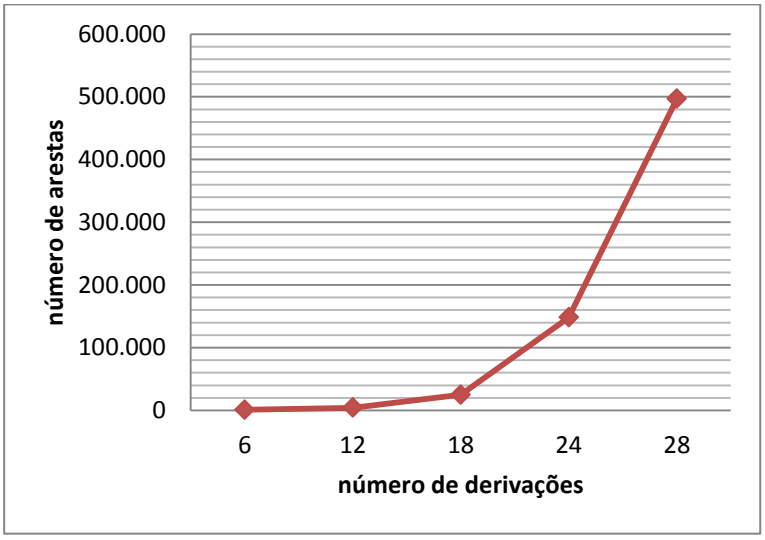

Figura 4.8 Número de derivações da malha secundária x Número de arestas (*rio de janeiro city*)

Novamente, nota-se que as curvas correspondentes à expansão da malha rodoviária primária crescem linearmente, enquanto que as curvas correspondentes à expansão da malha secundária crescem exponencialmente. Para entender a razão desses comportamentos é preciso analisar a complexidade algorítmica da expansão das malhas. Nesse algoritmo a entrada é o número de itens de trabalho. Pela descrição do procedimento de instanciação (Seção 3.2) deduz-se que o número de itens de trabalho cresce na ordem de três para um a cada intervalo fixo de derivações. Deduz-se, também, que todos os itens de trabalho produzem arestas no grafo, exceto aqueles que são

removidos pela aplicação das regras de restrição. Como a malha rodoviária primária cobre uma extensão maior do que a malha secundária, ela está mais sujeita à aplicação de regras de restrição. Isso explica porque a curva da expansão da malha primária comporta-se de maneira quase linear enquanto a curva da expansão da malha secundária comporta-se de maneira exponencial.

Coerentemente, a relação entre o número de derivações e o uso de memória assemelha-se à relação entre o número de derivações e o número de arestas (Figura 4.9 a Figura 4.12). Ressalta-se que os valores de uso de memória para a expansão da malha secundária incorporam os valores da expansão da malha primária. Adicionalmente, a expansão da malha primária inicia em um patamar de uso de memória mais elevado, pois cria referências às arestas na *quadtree*.

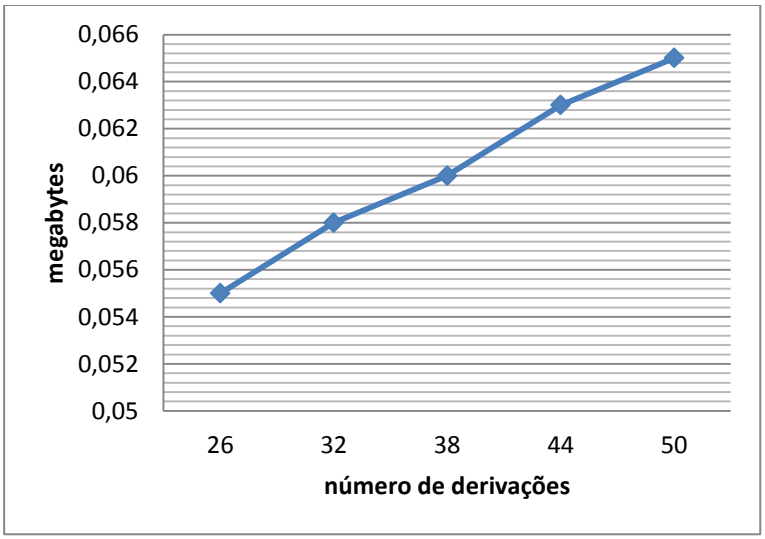

Figura 4.9 Número de derivações da malha primária x Uso de memória (*botafogo bay*)

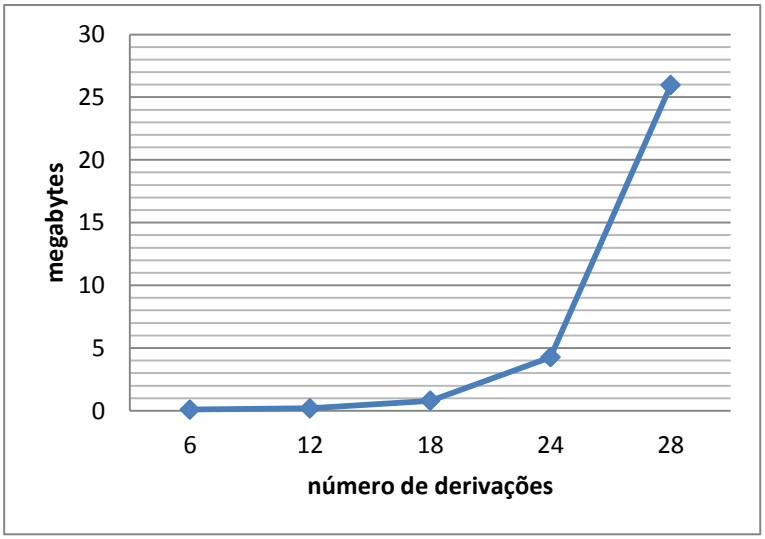

Figura 4.10 Número de derivações da malha secundária x Uso de memória (botafogo bay)

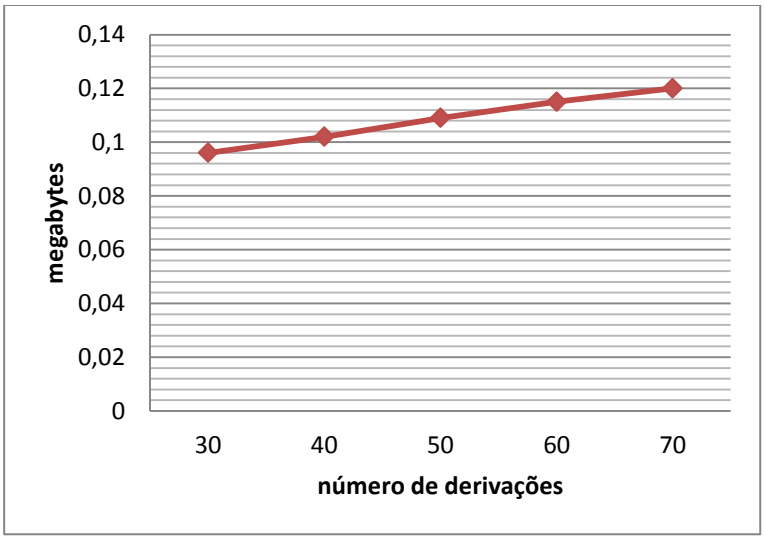

Figura 4.11 Número de derivações da malha primária x Uso de memória (*rio de janeiro city*)

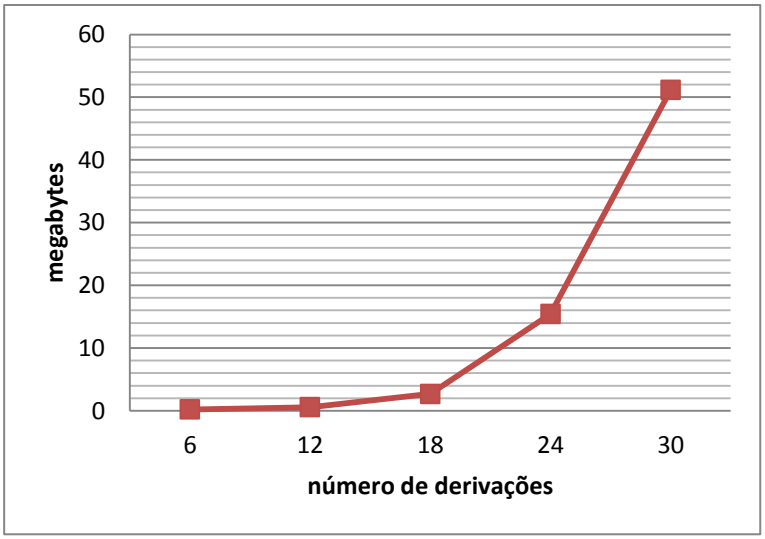

Figura 4.12 Número de derivações da malha secundária x Uso de memória (rio de janeiro city)

4.2. Impacto do aumento da granularidade da divisão espacial

No algoritmo proposto, a granularidade da divisão espacial tem influência direta no número de testes de colisão a serem realizados. Em uma *quadtree*, quanto mais níveis de profundidade, menor a área dos quadrantes, ou seja, mais dividido é o espaço. A *granularidade espacial ideal* é aquela que leva ao menor número de testes de colisão. Pode-se imaginar que a granularidade ideal é aquela que mais divide o espaço. No entanto, as características dos cenários desempenham um papel fundamental no número de testes de colisão, pois influenciam na distribuição das arestas pela área do mapa. Por exemplo, as zonas de bloqueio podem impedir a que a expansão avance por determinada direção, concentrando vias em partes do mapa. Por isso, é interessante observar experimentalmente a relação entre a divisão do espaço e o número de testes de colisão em cada cenário (Figura 4.13 e Figura 4.14) antes de prosseguir com outras análises sobre a divisão espacial.

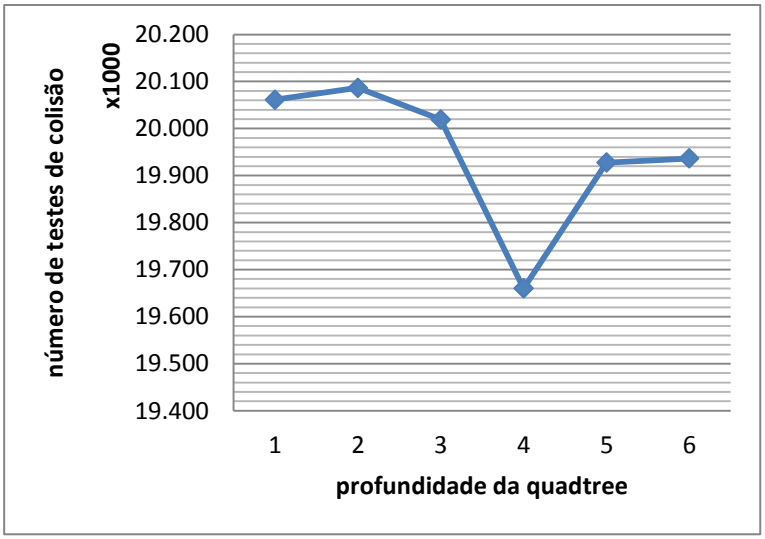

Figura 4.13 Profundidade da *quadtree* x Número de testes de colisão (*botafogo bay*)

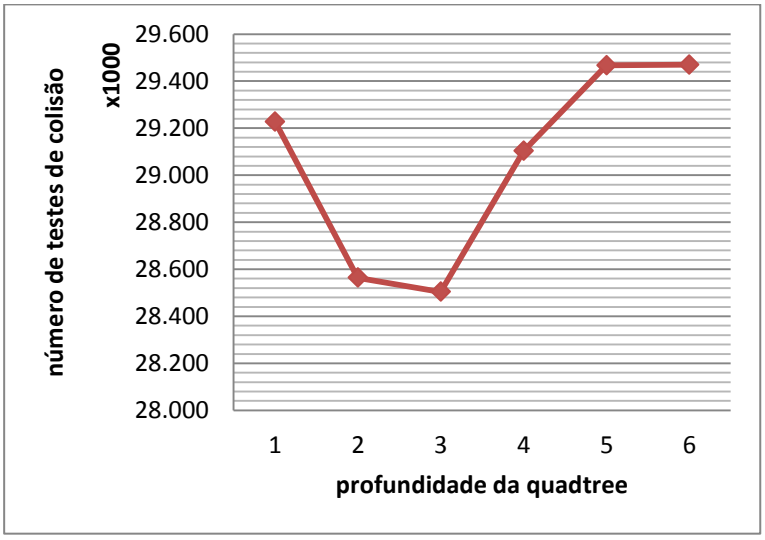

Figura 4.14 Profundidade da *quadtree* x Número de testes de colisão (*rio de janeiro city*)

Em cada curva pode-se notar o ponto onde o número de testes de colisão atinge o valor mínimo (granularidade espacial ideal). Nota-se, também, que a partir desse ponto, o número de testes de colisão tende a crescer. Atribui-se a esse fenômeno dois motivos. O primeiro diz respeito ao fato das arestas poderem ser referenciadas em mais de um quadrante. Desta forma, percebe-se que ao se aumentar a granularidade espacial sem diminuir o comprimento médio das arestas, aumenta-se a chance de haver mais referências a uma aresta em múltiplos quadrantes. O segundo motivo diz respeito à implementação da *quadtree* que divide automaticamente um quadrante em quatro sem usar uma

mínima de arestas na região. Com isso, o aumento da granularidade espacial aumenta a chance de que mais testes de colisão sejam realizados contra as caixas envolventes da hierarquia de um quadrante do que contra as arestas existentes no quadrante.

Naturalmente a granularidade espacial ideal também é observa da na análise do desempenho da expansão da malha primária (Figura 4.15 e Figura 4.17) e da detecção de colisão (Figura 4.16 e Figura 4.18).

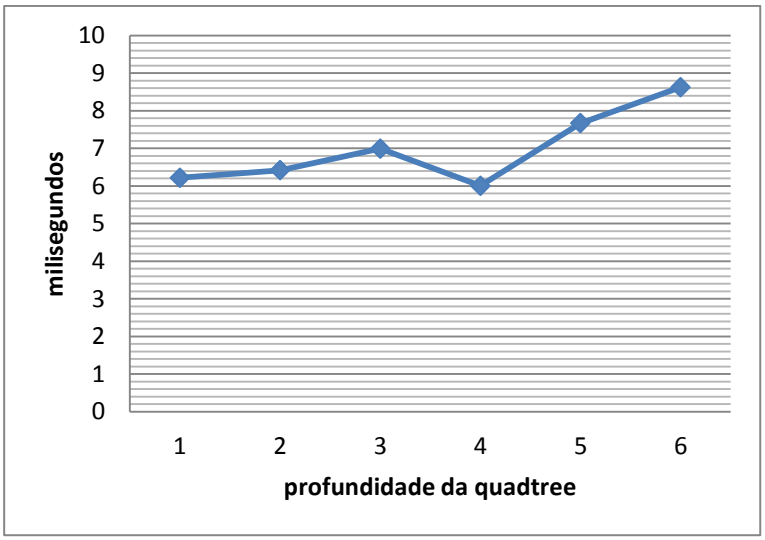

Figura 4.15 Profundidade da *quadtree* x tempo de execução da expansão da malha primária (*botafogo bay*)

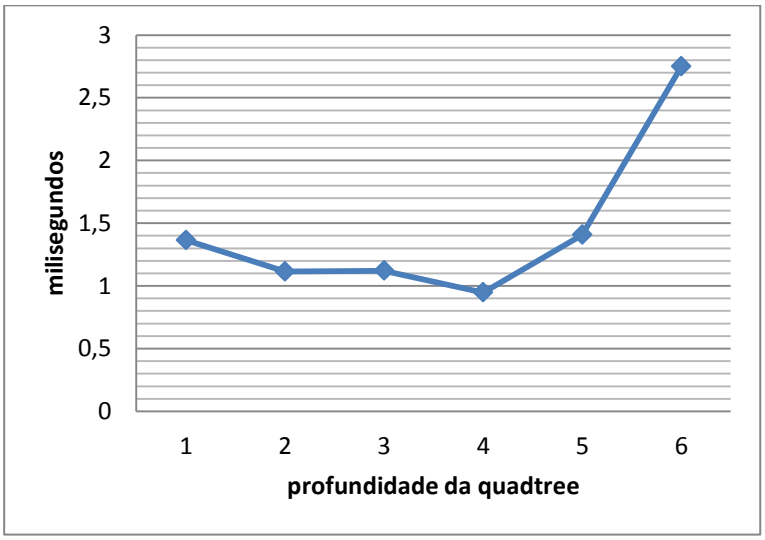

Figura 4.16 Profundidade da *quadtree* x tempo de execução da detecção de colisão (*botafogo bay*)

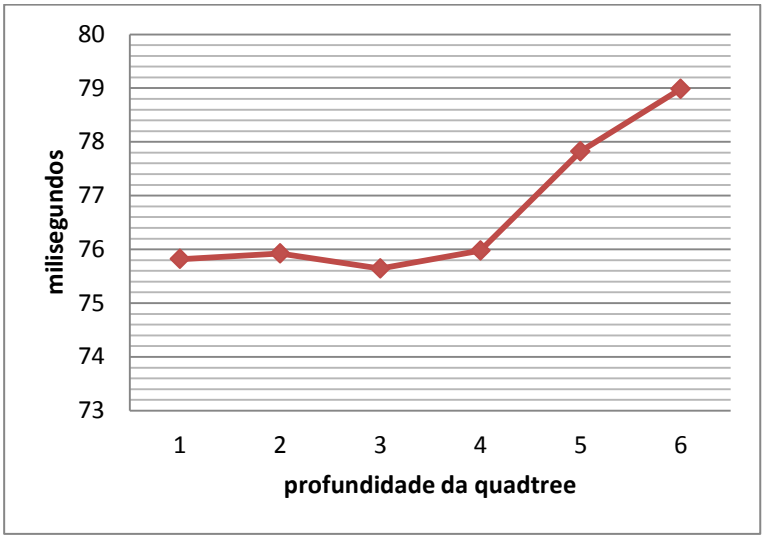

Figura 4.17 Profundidade da *quadtree* x tempo de execução da expansão da malha primária (rio de janeiro city)

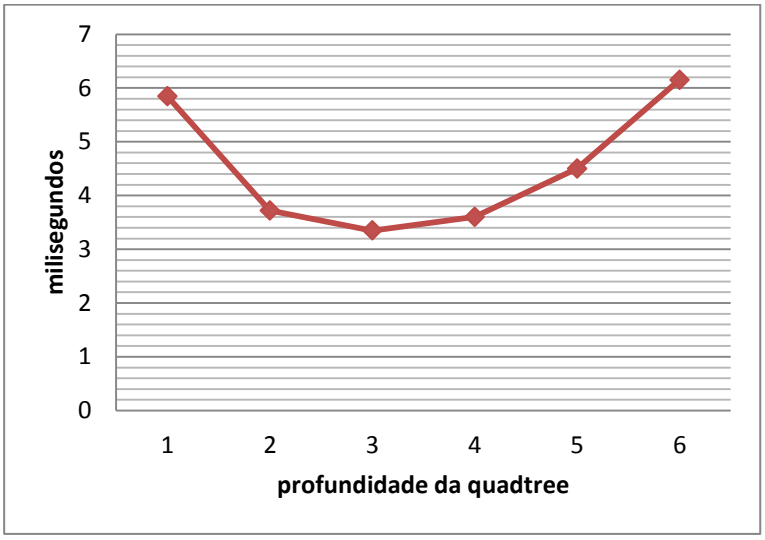

Figura 4.18 Profundidade da *quadtree* x tempo de execução da detecção de colisão (*rio de janeiro city*)

Na expansão da malha primária, atribui-se a melhoria de desempenho no ponto da granularidade ideal ao balanço entre o número de testes de colisão contra caixas envolventes e arestas. Na detecção de colisão, atribui-se a melhoria de desempenho ao balanço entre a área dos quadrantes e o comprimento médio das arestas.

4.3.
Impacto do aumento do paralelismo

A resposta de um algoritmo ao aumento do paralelismo depende, dentre outras coisas, da taxa de ocupação dos multiprocessadores gráficos que executam o algoritmo. A taxa de ocupação, por sua vez, depende da distribuição dos registradores e memória compartilhada entre as *threads* que executarão em um multiprocessador[5]. Se o algoritmo permitir a distribuição dos registradores e memória compartilhada entre muitas *threads*, é possível que ele tenha uma alta taxa de ocupação. Se um algoritmo for *bandwidth-bound*, como se suspeita que o algoritmo proposto seja, uma alta taxa de ocupação é desejável caso haja banda de memória suficiente a sua disposição [32][18]. Se um algoritmo *bandwitdh-bound* apresentar bom desempenho em alta taxa de ocupação, ele maneja bem a latência dos acessos à memória global.

Para analisar experimentalmente o impacto do aumento paralelismo na execução do algoritmo, computaram-se dois resultados diferentes. O primeiro relaciona o desempenho com a variação do número de blocos (Figura 4.19). O segundo relaciona o desempenho com a variação do número de *threads* (Figura 4.20).

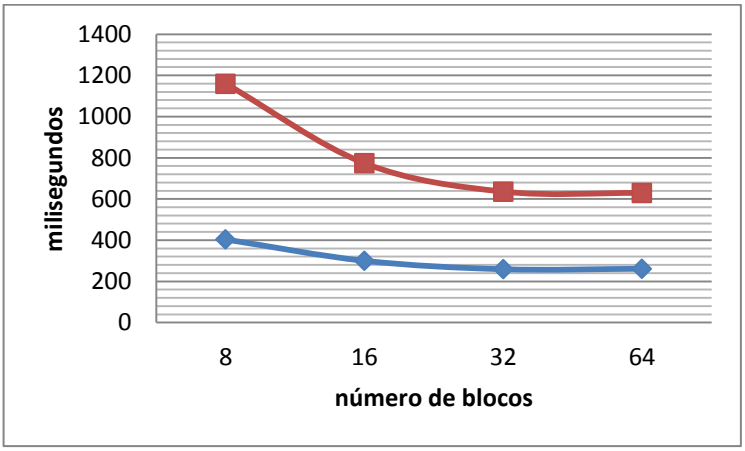

Figura 4.19 Número de blocos x Tempo de execução (*botafogo bay* em azul e *rio de janeiro city* em vermelho)

[5] Na arquitetura Kepler GK110, por exemplo, uma *thread* pode usar até 255 registradores e 48 KBytes de memória compartilhada privada.

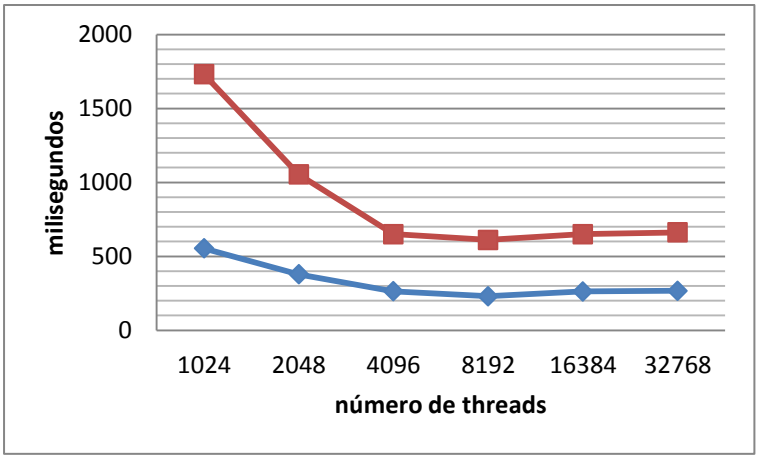

Figura 4.20 Número de *threads* x Tempo de execução (*botafogo bay* em azul e *rio de janeiro city* em vermelho)

Nota-se que o tamanho de bloco e número de *threads* com melhor desempenho são 32 e 256 respectivamente (8192 / 32 = 256 *threads* por bloco). No dispositivo usado para o teste há 4 multiprocessadores com capacidade de execução de até 8 blocos cada, totalizando 32 blocos ativos simultaneamente. Compilando o código CUDA com a diretiva "-Xptxas=-v", verificou-se que nenhuma função usa mais que 63 registradores. A especificação do dispositivo gráfico usado para o teste determina que, a um uso de 63 registradores por *thread*, a taxa de ocupação máxima é alcançada com 256 ou 512 *threads* por bloco [22]. Essa correspondência, além de confirmar a alta taxa de ocupação do algoritmo proposto, confirma que a latência no acesso a memória global está sendo bem manejada. Cria-se, então, uma expectativa de que o algoritmo apresente um desempenho ainda melhor em dispositivos que disponham de mais poder paralelo.

4.4. Impacto visual das mudanças no modelo de Parish et al.

Como visto na Seção 3.1.2, as condições de parada do algoritmo proposto não são as mesmas que no modelo de Parish et al. No modelo de Parish, controla-se a expansão das vias principais através da densidade populacional e da aderência. No algoritmo proposto, no entanto, controla-se a expansão das vias principais através do número de passos total da simulação e do número máximo de expansões para as vias principais.

Para observar o impacto do uso desses controles sobre o aspecto visual da malha primária, executou-se o algoritmo três vezes, com todos os parâmetros de entrada iguais, à exceção do controle sob análise.

Na Figura 4.21 observa-se as malhas rodoviárias resultantes do processo de geração com diferentes números de passos na simulação. A Figura 4.21a mostra uma malha resultante da simulação com 30 passos, enquanto a Figura 4.21b mostra a resultante de uma com 50 passos e a Figura 4.21c a resultante de uma com 70 passos.

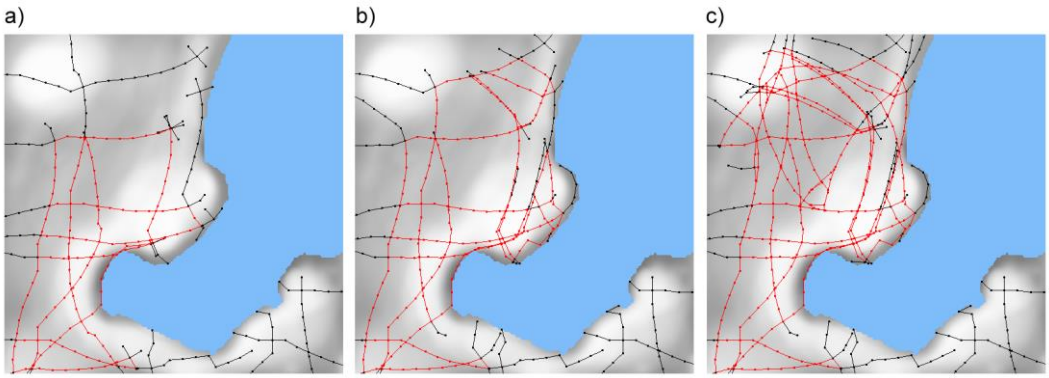

Figura 4.21 Malhas rodoviárias resultantes da geração com (a) 30 passos de simulação, (b) 50 passos de simulação e (c) 70 passos de simulação

Nota-se que o aumento no número de passos da simulação não só desempenha um papel importante na distribuição da malha rodoviária gerada, como também se relaciona com o primeiro efeito colateral indesejável das mudanças realizadas ao modelo de Parish et al.: a ocorrência de vias principais próximas e concorrentes. Crê-se que a ocorrência desse efeito colateral esteja relacionada ao fato de não se haver reduzido a densidade populacional durante a expansão da malha primária. Por causa disso, ao aumentar o número de passos da simulação, e consequentemente o número de vias principais, mais vias convergiram para os mesmos pontos com alta densidade populacional, desenvolvendo trajetórias paralelas.

Na Figura 4.22 observa-se as malhas rodoviárias resultantes do processo de geração com diferentes números de passos máximos de expansão das vias principais. A Figura 4.22a mostra uma malha resultante da geração com 4 passos máximos para a expansão das vias principais, enquanto que a Figura

4.22b mostra uma resultante da geração com 8 passos máximos e a Figura 4.22c mostra uma resultante da geração com 16 passos máximos.

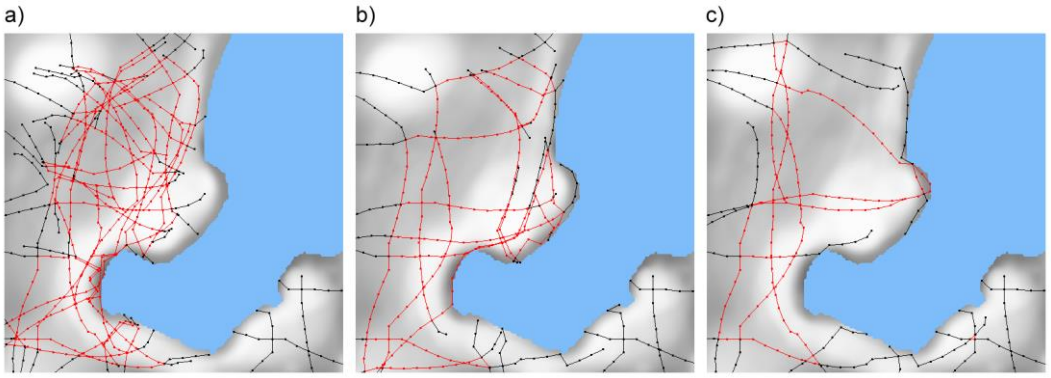

Figura 4.22 Malhas rodoviárias resultantes da geração com (a) 4 passos, (b) 8 passos e (c) 16 passos máximos para a expansão das vias principais

Nota-se que a redução do número máximo de expansões para vias principais desempenha um papel importante na quantidade e na concentração das vias principais entre os pontos de maior densidade populacional. Além disso, a redução do número máximo de expansões também deixa mais evidente um segundo efeito colateral das mudanças realizadas ao modelo de Parish: a formação de células rodoviárias muito estreitas ou muito largas. Crê-se que a ocorrência desse efeito colateral esteja relacionada ao fato de não se haver interrompido a expansão das vias principais quando elas se cruzam, somado ao fato das vias tenderem a realizar trajetórias paralelas, como explicado anteriormente.

No entanto, através de um processo experimental, descobriu-se que o ajuste dos controles (número de passos da simulação e do número máximo de expansões das vias principais), juntamente com o ajuste dos pontos iniciais para a expansão da malha primária, pode minimizar a ocorrência dos efeitos colaterais oriundos das mudanças no modelo de Parish et al. (Figura 4.23).

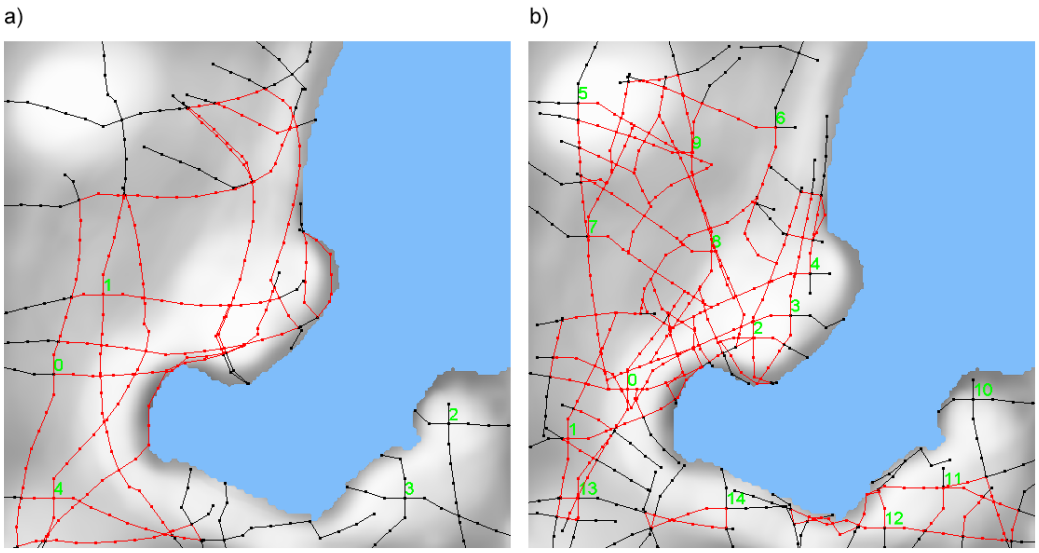

Figura 4.23 Malhas rodoviárias resultantes da geração com (a) 50 passos, 8 expansões máximas e 5 pontos iniciais, e (b) 16 passos, 6 expansões máximas e 15 pontos iniciais

A Figura 4.23a mostra a malha rodoviária resultante da geração com 50 passos de simulação, 8 expansões máximas para cada via principal e 5 pontos iniciais de expansão. Já a Figura 4.23b mostra a malha resultante da geração com 16 passos de simulação, 6 expansões máximas e 15 pontos iniciais. Na segunda figura é possível notar a diminuição da ocorrência de vias próximas e paralelas e o aumento da formação de células rodoviárias proporcionais.

4.5. Comparação entre as execuções na CPU e na GPU

Para comparar as execuções na GPU e CPU, executou-se o algoritmo em duas configurações de hardware (Tabela 4.1). Pela ocasião em que esse trabalho foi escrito, a primeira configuração representa uma opção de médio custo (*mid-end*) e a segunda uma opção de alto custo (*high-end*).

Tabela 4.1 Configurações de hardware onde o algoritmo proposto foi executado

Processador	Placa Gráfica
Intel® Core™ i7-4930MX	NVIDIA GeForce GTX 750M

Intel® Core™ i7-3770	NVIDIA GeForce GTX Titan

A Figura 4.24 e a Figura 4.25 mostram a relação entre o tempo e o número de arestas nas diferentes configurações da Tabela 4.1. Para permitir a comparação entre as diferentes configurações, a Figura 4.26 e a Figura 4.27 mostram a mesma informação, mas com o tempo normalizado para cada grupo de número de arestas. As barras azul, vermelha e verde correspondem, respectivamente, ao processador i7-4930MX (processador com o melhor desempenho) e as placas gráficas GTX 750M e GTX Titan.

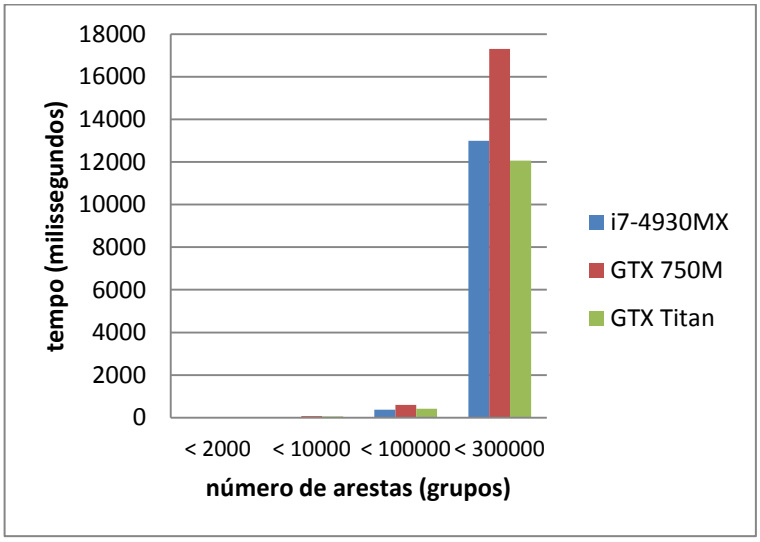

Figura 4.24 Comparação entre os tempos de execução na CPU e nas duas GPUs (botafogo bay)

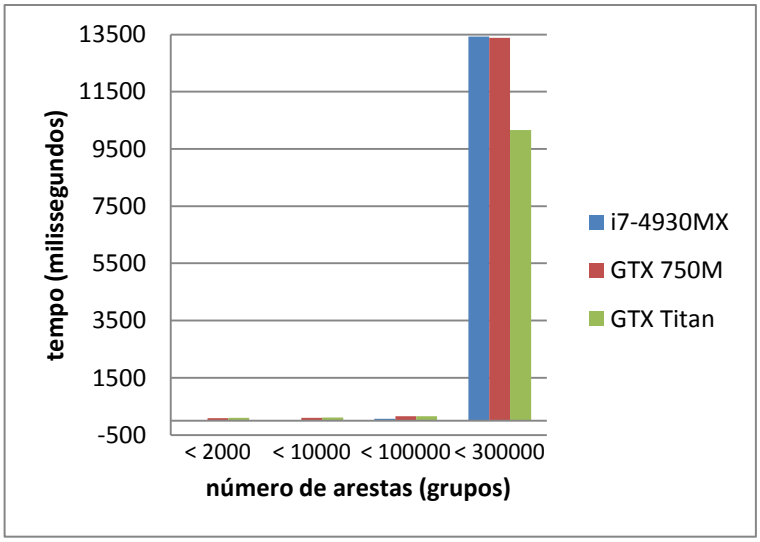

Figura 4.25 Comparação entre os tempos de execução na CPU e nas duas GPUs (rio de janeiro city)

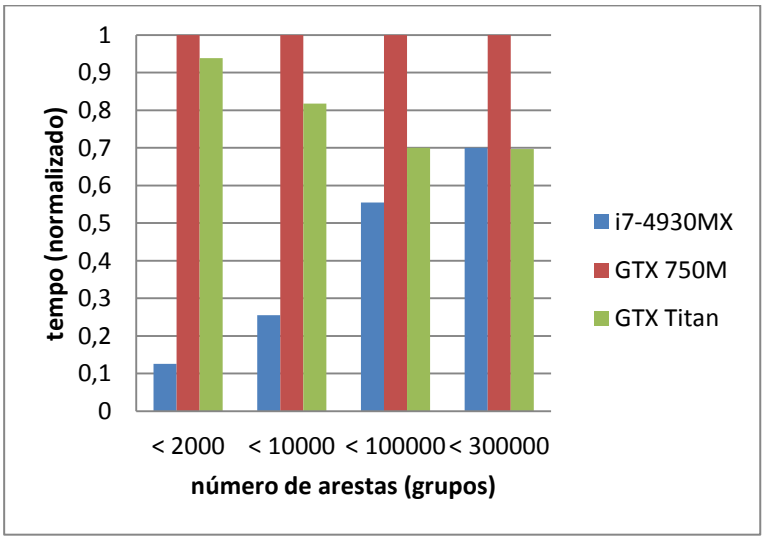

Figura 4.26 Comparação entre os tempos de execução (normalizados) na CPU e nas duas GPUs (*botafogo bay*)

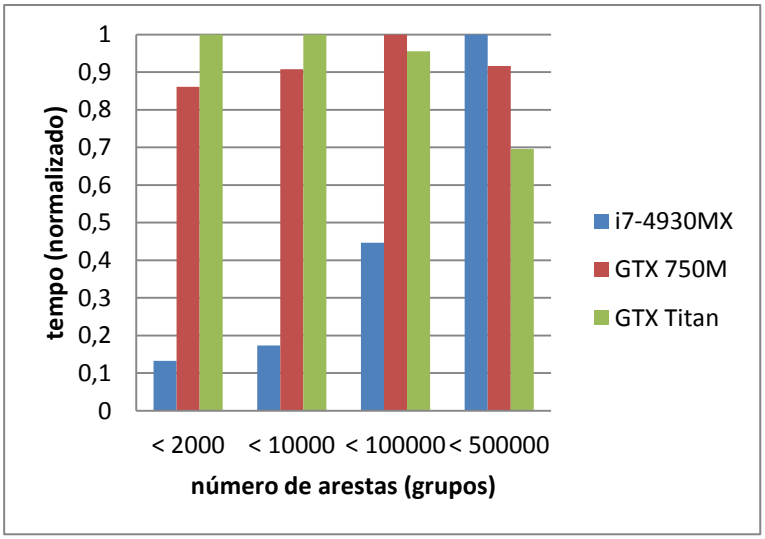

Figura 4.27 Comparação entre os tempos de execução (normalizados) na CPU e nas duas GPUs (*rio de janeiro city*)

Nota-se que a execução do algoritmo na CPU supera a execução do algoritmo na GPU (ambas as configurações) quase sempre, à exceção de quando número de arestas produzidas é grande (> 300.000). Além disso, o desempenho superior na GTX Titan confirma a expectativa da Seção 4.3 sobre a melhoria de desempenho em dispositivos com maior poder paralelo.

5
Conclusões

Este trabalho apresenta um algoritmo original no sentido de ser o primeiro a explorar a geração de malhas rodoviárias na GPU. Para elaborá-lo, adotou-se o modelo de Parish et al. A fim de permitir a execução do algoritmo proposto nos multiprocessadores gráficos, modificou-se o modelo de Parish em alguns pontos. Adotou-se, também, CUDA como tecnologia de implementação na GPU. Devido a inexistência de dados de desempenho nos trabalhos relacionados, implementou-se o algoritmo proposto também na CPU. Por fim, analisou-se o impacto das decisões tomadas sobre o aspecto das malhas rodoviárias geradas e o desempenho da execução do algoritmo na GPU, comparando-o com o desempenho na CPU.

Como é possível notar na Seção 4.4, as alterações ao modelo de Parish fazem que o algoritmo requeira o ajuste experimental de alguns parâmetros de entrada a fim de gerar malhas rodoviárias mais interessantes. Aliado a isso, a escolha por não implementar todos os padrões urbanísticos e por não permitir a mistura de padrões restringiram a capacidade de representação de malhas rodoviárias geradas pelo algoritmo. Ademais, acredita-se que o baixo desempenho ao gerar malhas rodoviárias com poucas arestas na GPU indica uma baixa carga de trabalho inicial, principalmente na expansão da malha rodoviária primária. Com poucas vias principais a expandir não é possível tirar real vantagem da computação paralela. Apenas nas iterações tardias da geração da malha primária, quando a carga de trabalho se torna significativa, justifica-se sua execução na GPU. O desempenho abaixo do esperado, apesar do cuidado com a latência no acesso à memória, indica uma baixa intensidade aritmética. Ter uma baixa intensidade aritmética significa ter uma baixa razão entre o número de operações aritméticas e o número de operações de carregamento e gravação de memória. Esse desequilíbrio minimiza a vantagem da arquitetura *throughput-oriented* em executar um número muito maior de operações de ponto flutuante por segundo (GFLOP/s) em detrimento de uma estratégia de cache menos poderosa [23].

No entanto, pelo caráter exploratório deste trabalho, crê-se que a identificação dos problemas mencionados anteriormente represente uma

contribuição ao tema. Através da análise da execução do algoritmo proposto na GPU, identificou-se questões importantes, como a necessidade de alinhar o paradigma da amplificação de dados [8] com a necessidade de elevada carga de trabalho inicial para fazer bom uso das GPUs. Alguns desafios de implementação também apontam para questões que vão além do modelo adotado, como por exemplo, a necessidade do uso de *locks* na detecção de colisão, indicativo de uma dependência de ordem intrínseca à geração por expansão. Além disso, a elaboração de um algoritmo com estrutura modular, baseado em filas de trabalho, permite a adição de novos padrões urbanísticos e novas funcionalidades de maneira mais intuitiva.

Assim, a despeito de não ter culminado em uma solução definitiva para a geração de malhas rodoviárias na GPU, espera-se que os resultados e conclusões apresentados sirvam de ponto partida para futuros trabalhos no tema.

5.1. Trabalhos Futuros

Imagina-se que novas investigações sobre o tema enderecem as questões deixadas em aberto, tanto aquelas sobre a capacidade de representação de malhas rodoviárias quanto aquelas sobre desempenho. Sabe-se, no entanto, que trabalhos futuros podem endereçar essas questões de duas maneiras: através da elaboração de um algoritmo novo ou através de modificações ao algoritmo proposto. Assim, esta seção apresentará recomendações, que podem ser usadas na elaboração de um algoritmo novo, e sugestões, que podem ser usadas em modificações ao algoritmo proposto.

Para endereçar as questões sobre desempenho recomenda-se, primeiramente, gerar uma carga de trabalho inicial suficientemente grande de forma a aumentar o uso precoce do poder computacional paralelo das GPUs. Para isso sugere-se estudar a possibilidade de pré-computar múltiplos pontos iniciais para a expansão da malha primária (*spawn_points*) a partir da extração dos pontos de maior concentração populacional no mapa de densidade populacional. Adicionalmente, recomenda-se usar outro tipo de representação das informações contidas nos mapas de imagem a fim de aumentar a intensidade aritmética do algoritmo. Sugere-se, então, estudar a possibilidade de pré-computar polígonos a partir dos corpos d'água e zonas de bloqueio para que

a aplicação das regras de restrição seja feita através de testes de colisão contra estes polígonos ao invés da amostragens nos mapas de imagem.

Para endereçar as questões de capacidade de representação de malhas rodoviárias recomenda-se, primeiramente, elaborar um mecanismo que leve em consideração a alteração da densidade populacional. Para isso sugere-se estudar a aplicação de um mecanismo análogo ao de transferência de calor, cuja simulação paralela nas GPUs já fora explorada em outras literaturas [29], e cujo efeito atende ao anseio de conduzir a expansão para áreas não visitadas. Ademais, recomenda-se implementar outros padrões urbanísticos e a mistura de padrões. Note-se que atualmente a expansão da malha rodoviária secundária não testa a detecção de colisão entre vias locais por causa da previsibilidade do padrão *checkerboard*. Presume-se que novos padrões urbanísticos requererão detecção de colisão entre vias locais, ainda que essa colisão não provoque a interrupção da expansão.

Referências Bibliográficas

[1] ALEXANDER, C; ISHIKAWA, S; SILVERSTEIN, M; **A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)**; Editora da Universidade de Oxford, Nova Iorque, EUA, 1977.

[2] ASHLOCK, D; BRYDEN, K; GENT, S; **Evolution of L-systems for Compact Virtual Landscape Generation**; IEEE Congress on Evolutionary Computation, 2005.

[3] CEDERMAN, D; TSIGAS, P; **On Dynamic Load Balancing on Graphics Processors**; Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pgs. 57-64, 2008.

[4] CHEN, G; ESCH, G; MÜLLER, P; WONKA, P; ZHANG, E; **Interactive Procedural Street Modeling**; ACM TRANS. ON GRAPHICS; 2008.

[5] CHOMSKY, N; **Three models for the description of language**; IRE Trans. on Information Theory, vol. 2(3), pgs. 113-124, 1956.

[6] DIETRICH, K; ROTACH, M; BOPPART, E; **Strassenprojektierung (Road Project)**; Zurique, 1993.

[7] EBERLY, D; **The Minimal Cycle Basis for a Planar Graph**; Geometric Tools, Inc., 2005.

[8] EBERT, D; MUSGRAVE, F; PEACHEY, D; PERLIN, K; WORLEY, S; **Texturing and Modeling: A Procedural Approach**; Ed. Morgan Kaufmann, 2002.

[9] FARBER, R; **CUDA Application Design and Development**; Ed. Morgan Kaufman, 2013.

[10] FUESSER, K; **Strasse & Verkehr (City Roads and Traffic)**; Ed. Vieweg, 1997.

[11] GROSS, M; PARISH, Y; **Design und Implementation Einer Preprocessing Pipeline zur Visualisierung Prozedural Erzeugter Stadtmodelle (Design and Implementation of a Preprocessing Pipeline for**

Visualizing Procedurally Generated City Models); Dissertação de Mestrado, Instituto Federal de Tecnologia de Zurique, 2001.

[12] GUPTA, K; OWENS, J; STUART, J; **A study of persistent threads style GPU programming for GPGPU workloads**; Innovative Parallel Computing, vol. 14, 2012.

[13] HWU, W; Kirk, D; **Programming Massively Parallel Processors - a Hands-on Approach**; Ed. Morgan Kaufman, 2013.

[14] LECHNER, T; WATSON, B; WILENSKY, U; FELSEN, M;**Procedural City Modeling**; Universidade Northwestern, Illinóis, EUA, 2003.

[15] LINDENMAYER, L; PRUSINKIEWICZ, P; **The Algorithmic Beauty of Plants**; Ed. Springer, 1990.

[16] LIPP, M; WIMMER, M; WONKA, P; **Parallel Generation of L-Systems**; VMV - Vision, Modeling and Visualization '09, pgs. 205-214, Brunsvique, Alemanha, 2009.

[17] LIPP, M; WIMMER, M; WONKA, P; **Parallel Generation of Multiple L-Systems**; Computers & Graphics, Nº. 34(5), pgs. 585-593, 2010.

[18] LUITJENS, J; RENNICH, S; **CUDA Warps and Occupancy**; CPU Computing Webinar, 2011.

[19] LYNCH K; **The Image of the City**; Ed. do Instituto de Tecnologia de Massachusetts, EUA, 1960.

[20] MANOCHA, D; MERRELL, P; **ModelSynthesis - A General Procedural Modeling Algorithm**; Universidade de Stanford, Califórnia, EUA, 2010.

[21] MANOUSAKIS, S; **Musical L-Systems**; Dissertação de Mestrado, Conservatório Real de Haia, Haia, Holanda, 2006.

[22] NVIDIA; **CUDA Occupancy Calculator**; 2012.

[23] NVIDIA; **CUDA Programming Guide**; 2001.

[24] OLSEN, J; **Realtime Procedural Terrain Generation**; Relatório Técnico, Universidade do Sul da Dinamarca, Dinamarca, 2004.

[25] PARISH, Y.; MÜLLER, P.; **Procedural Modeling of Cities**; SIGGRAPH '01 Proceedings; pgs. 301-308, 2001.

[26] PERLIN, K; **Making Noise**; http://www.noisemachine.com/talk1/index.html, 1992, Acessado em: 28/02/2014.

[27] PRESS, W; TEUKOLSKY, S; VETTERLING, W; FLANNERY, B; **Numerical recipes in C: The art of scientific computing**; Ed. Universidade de Cambridge, Nova Iorque, EUA, 1988.

[28] PRUSINKIEWICZ, P; JAMES, M; MECH, R; **Synthetic Topiary**; Proceedings of SIGGRAPH '94, pgs. 351-358, Florida, EUA, 1994.

[29] SANDERS, J; KANDROT, E; **CUDA By Example: Na Introduction to General-Purpose GPU Programming**; Ed. Addison-Wesley Professional, 2010.

[30] STEINBERGER, M; KAINZ, B; KERBL, B; HOUSWIESNER, S; KENZEL, M; SCHMALSTIEG, D; **Softshell: Dynamic Scheduling on GPUs**; Proceedings of ACM SIGGRAPH Asia 2012, Volume 31, Exemplar 6, Art. Nº. 161, 2012.

[31] SUN, J; XIAOBO, Y; BACIU, G; GREEN, M; **Template-based Generation of Road Networks for Virtual City Modeling**; VRST-02, Ed. ACM, pgs. 33-40, 2002.

[32] VOLKOV, V; **Better Performance at Lower Occupancy**; GPU Technology Conference, 2010.

Apêndice A

A.1 - Tabela de Parâmetros

Parâmetro	Descrição	Tipo/Unidade
seed	Semente do gerador de números pseudo-aleatórios	Inteiro
name	Nome da cidade	Literal
world_[width/height]	Largura/Altura da cidade	Inteiro/Pixels
population_density_map	Caminho para o mapa de densidade populacional	Literal
water_bodies_map	Caminho para o mapa de corpos d'água	Literal
blockades_map	Caminho para o mapa de zonas bloqueios	Literal
[natural/raster/radial]_pattern_map	Caminho para o mapas de zonas de padrão	Literal
spawn_points	Pontos para início da simulação	Vetor de pontos no R^2 na forma $[(x_1,y_1,z_1), \ldots, (x_n, y_n, z_n)]$
max_[highway/street]_derivation	Número máximo de iterações (derivações) para expansão da via principal/local	Inteiro
[highway/street]_length	Comprimento inicial	Ponto

	de uma via principal/local	flutuante/Pixels
max_highway_goal_deviation	Ângulo máximo de desvio de uma via principal	Ponto flutuante/Graus
highway_branching_distance	Distância para a criação de ramificações em uma via principal	Inteiro/Nº. de derivações
max_[highway/street]_branch_depth	Número máximo de gerações de uma ramificação de uma via principal/local	Inteiro/Nº. de derivações
street_branching_delay	Retardo para início da ramificação de uma via local	Inteiro/Nº. de derivações
max_obstacle_deviation_angle	Ângulo máximo de desvio de um obstáculo	Ponto flutuante/Graus
min_road_length	Comprimento mínimo de uma via ao desviar de um obstáculo	Inteiro/Pixels
min_block_area	Tamanho mínimo de uma célula rodoviária	Inteiro/Área em pixels
sampling_arc	Ângulo do arco de amostragem no mapa de densidade populacional	Ponto flutuante/Graus
[min/max]_sampling_ray_length	Comprimento mínimo/máximo do raio de amostragem no mapa de densidade populacional	Inteiro/Pixels
min_sampling_weight	Peso mínimo requerido de uma	Inteiro

	amostragem no mapa de densidade populacional	
goal_distance_threshold	Tolerância de distância para o alcance de um objetivo por uma via principal	Ponto flutuante/Pixels
quadtree_depth	Profundidade máxima da Quadtree	Inteiro
snap_radius	Distância mínima entre vértices para que haja aderência	Ponto flutuante/Pixels
draw_spawn_point_labels	Habilita/desabilita o desenho de rótulos nos pontos de início de expansão	Booleano
draw_graph_labels	Habilita/desabilita o desenho de rótulos nas arestas e vértices do grafo da malha	Booleano
cycle_color	Cor das vias principais que compõem células rodoviárias	Cor na forma (r, g, b, a)
filament_color	Cor das vias principais que não compõem células rodoviárias	Cor na forma (r, g, b, a)
street_color	Cor das vias locais	Cor na forma (r, g, b, a)
label_font_size	Tamanho de renderização da letra dos rótulos	Ponto flutuante
point_size	Tamanho de	Ponto flutuante

	renderização de um vértice do grafo	
max_vertices	Número máximo de vértices do grafo	Inteiro
max_edges	Número máximo de arestas do grafo	Inteiro
max_primitives	Número máximo de células rodoviárias	Inteiro
vertex_buffer_size	Número máximo de vértices na placa gráfica	Inteiro
index_buffer_size	Número máximo de índices na placa gráfica	Inteiro