Pontifícia Universidade Católica
do Rio de Janeiro

Wallas Henrique Sousa dos Santos

MCAD Shape Grammar: Modelagem procedimental em modelos CAD massivos industriais

Tese de Doutorado

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática da PUC-Rio.

Orientador: Prof. Alberto Barbosa Raposo

Rio de Janeiro
Abril de 2018

Pontifícia Universidade Católica
do Rio de Janeiro

Wallas Henrique Sousa dos Santos

MCAD Shape Grammar: Modelagem procedimental em modelos CAD massivos industriais

Tese apresentada como requisito parcial para obtenção do grau de Doutor 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. Bruno Feijó
Departamento de Informática – PUC-Rio

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

Prof. Cesar Tadeu Pozzer
Universidade Federal de Santa Maria – UFSM

Dr. Emilio Ashton Vital Brazil
IBM Research – Brazil

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 20 de Abril de 2018

Wallas Henrique Sousa dos Santos

O autor possui Bacharelado (2011) e Mestrado (2013) em Ciências da Computação pela Universidade Federal do Maranhão, no qual foi pesquisador do Núcleo de Computação Aplicada (NCA). Desde 2014, durante o Doutorado, foi desenvolvedor e pesquisador pelo Tecgraf trabalhando com modelos CAD massivos até 2016. Atualmente é estagiário de Doutorado da IBM Research | Brazil onde tem contribuído para o desenvolvimento de novas abordagens de especificar representações híbridas de conhecimento.

Ficha Catalográfica

Santos, Wallas Henrique Sousa dos

MCAD Shape Grammar: Modelagem procedimental em modelos CAD massivos industriais / Wallas Henrique Sousa dos Santos; orientador: Alberto Barbosa Raposo. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2018.

v., 112 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Modelos CAD 3D;. 3. Modelos CAD Massivos;. 4. Shape Grammars;. 5. Rendering em tempo real.. I. Raposo, Alberto Barbosa. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Por minha família

Agradecimentos

Agradeço primeiro a minha família Vera, João, Caio e Rafaela pelo amor e apoio, sem eles não estaria aqui.

Agradeço aos professores da PUC-Rio, em especial, Alberto Raposo por me aceitar, orientar e ter me ajudado muito durante minha trajetória do doutorado, Marcelo Gattass por me receber no início do curso e me integrado ao Tecgraf.

Agradeço aos amigos do Tecgraf, Eduardo Tadeu por ter sido meu gerente e me receber no Environ, Paulo Ivson que me ensinou quase tudo que precisei na minha pesquisa, Eduardo Telles e Thales Sabino por terem me incentivado e ajudado a programar melhor. Ao pessoal do grupo do Alberto também que me cederem o espaço e equipamento para escrever a tese na reta final.

Agradeço a todos os meus amigos, em especial André Moreira, Suellen Motta, Lucas Caracas, Polyana Sá, Rodrigo Costa, André Reis, Marcela Câmara, Thaysa Cunha e Natália Moraes que conviveram ou convivem comigo e me deram todo apoio e incentivo em diversos momentos dessa trajetória.

Resumo

Santos, Wallas Henrique Sousa dos; Raposo, Alberto Barbosa. **MCAD Shape Grammar: Modelagem procedimental em modelos CAD massivos industriais**. Rio de Janeiro, 2018. 112p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Modelos CAD 3D são ferramentas utilizadas na indústria para planejamento e simulações antes da construção ou realização de tarefas. Em muitos casos, como por exemplo na indústria de óleo e gás, esses modelos podem ser massivos, ou seja, possuem informações detalhadas em larga escala no intuito de que sejam fontes de informações precisas. Para obtenção de navegação interativa nesses modelos é necessária uma combinação de hardware e software adequados. Mesmo hoje com GPUs mais modernas, a renderização direta desses modelos não é eficiente, sendo necessárias abordagens clássicas como descarte de objetos não visíveis e LOD antes de enviar os dados à GPU. Logo, para renderização em tempo real de modelos CAD massivos são necessários algoritmos e estruturas de dados escaláveis para processamento da cena de forma eficiente. O trabalho dessa tese propõe o MCAD (*Massive Computer-Aided Design*) *Shape grammar*, uma gramática expansiva que gera objetos para criar cenas 3D de modelos massivos de forma procedimental. Nos últimos anos, modelagem procedimental tem ganhado atenção para criar cenas 3D rapidamente utilizando uma representação compacta, que armazena regras de geração ao invés de representação explícita da cena. MCAD *Shape grammar* explora repetições e padrões presentes em modelos massivos para renderização de cenas, reduzindo o consumo de memória e processando a cena procedimentalmente de forma eficiente. Convertemos modelos reais de refinarias em MCAD *Shape grammar* e implementamos um renderizador para os mesmos. Os resultados mostraram que esta solução é escalável com alto desempenho, além de ser a primeira vez que modelagem procedimental é utilizada nesse domínio.

Palavras-chave

Modelos CAD 3D; Modelos CAD Massivos; Shape Grammars; Rendering em tempo real.

Abstract

Santos, Wallas Henrique Sousa dos; Raposo, Alberto Barbosa (Advisor). **MCAD Shape Grammar: Procedural Modeling for industrial Massive CAD Models**. Rio de Janeiro, 2018. 112p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

3D CAD models are tools used in the industry for planning and simulations before construction or completion of tasks. In many cases, such as in the oil and gas industry, these models can be massive, that is, they have large-scale detailed information in order to be sources of accurate information. Interactive navigation in these models requires a combination of appropriate hardware and software. Even nowadays with modern GPUs, the direct rendering of these models is not efficient, requiring classic approaches such as culling non-visible objects and LOD before sending the data to the GPU. Therefore, for real-time rendering of massive CAD models, we need scalable algorithms and data structures to efficiently process the scene. The work of this thesis proposes MCAD (Massive Computer-Aided Design) Shape grammar, an expansive grammar that procedurally generates objects to create 3D scenes of massive models. In recent years procedural modeling has drawn attention for quickly creating 3D scenes using a compact representation, which stores generation rules rather than explicit representation of the scene. MCAD Shape grammar explores repetitions and patterns present in massive models for rendering scenes, reducing memory footprint and procedurally processing the scene efficiently. We converted real refinery models into MCAD Shape grammar and implemented a renderer for them. Results showed that our solution is scalable with high performance, also it is the first time that procedural modeling is used in this domain.

Keywords

3D CAD Models; Massive CAD Models; Shape Grammars; Real-time rendering.

Sumário

Lista de figuras

Lista de tabelas

Lista de Abreviaturas

AABB – *Axis-aligned bounding box* (Caixa envolvente alinhada com os eixos)
API – *Application Programming Interface*
BIM – *Building Information Modeling*
BSP – *Binary Space Partitioning*
BVH – *Bounding Volume Hierarchy*
CAD – *Computer Aided Design*
CPU – *Central Process Unit*
FPS – *Frames per second* (Quadros por segundo)
GPU – *Graphics Processing Unit*
IFC – *Industry Foundation Classes*
LOD – *Level of detail*
NURBS – *Non-Uniform Rational Basis Spline*
OBB – *Oriented bounding box* (Caixa envolvente orientada com os eixos)
MCAD – *Massive Computer-Aided Design*
PDMS – *Plant Design Management System*
SIMD – *Single Instruction Multiple Data*
SSAO – *Screen Space Ambient Occlusion*
RA – Realidade Aumentada
RV – Realidade Virtual

Computação gráfica!? É tudo mentira!

Thales Sabino, *Tecgraf 2014.*

1
Introdução

Modelos CAD 3D são ferramentas importantes para planejamento, manutenção e operação de projetos industriais. Um modelo CAD, usualmente, deve ser fidedigno e preciso para ser fonte de informação para análise e simulação. BIM (*Business information modeling*) é um processo de gerenciamento de projetos industriais amplamente utilizado pela engenharia que visa sistematizar a comunicação entre as partes envolvidas utilizando dados digitais, principalmente através de modelos CAD [1]. Exemplos de caso de uso incluem estudo de viabilidade [2], detecção de colisão [3], simulações [4] e treinamento [5] que podem ser utilizados para minimização de custos, otimização de processos, verificação de segurança e impactos ambientais. Os engenheiros de um projeto estão interessados em analisar modelos em vários níveis de escala de modelo, como por exemplo, um equipamento individualmente, ou o conjunto no qual faz parte assim como a relação com demais estruturas do projeto. Esse trabalho referencia modelos CAD como modelos 3D massivos para plantas industriais, porém as análises podem ser aplicadas em domínios semelhantes.

Por definição, modelos são representações simplificadas de uma entidade real. Entretanto, projetos de grande escala como de plantas industriais geram modelos altamente detalhados que podem ser complexos de modelar e/ou navegar mesmo em computadores modernos. Plantas industriais podem conter milhões de objetos geométricos. Dessa forma, sistemas de visualização podem enfrentar problemas de escalabilidade sem o conjunto de *hardware* e algoritmos adequados. Além disso, no âmbito da modelagem, modelos com defeitos de modelagem ou incompletos podem causar impacto negativo no planejamento de tarefas, logo podem causar prejuízos financeiros ou mesmo acidentes. Esses desafios motivam a pesquisa para novas soluções no domínio de CAD.

A abordagem direta ou por força bruta de renderizar uma cena 3D, em geral, não é eficiente mesmo utilizando GPU modernas para modelos CAD massivos. Dada a magnitude desses modelos, é necessário pré-processar a representação do modelo, seja antes do carregamento de modelo pelo sistema ou durante a execução antes de enviar os dados para GPU na forma de triângulos ou qualquer outra representação específica que o sistema de visualização tenha suporte. Dentre as abordagens clássicas, há o descarte de objetos

baseado em visibilidade como *frustum culling* e *occlusion culling* [6, 7] e a alternância do nível de detalhe na renderização desenhando simplificações de objetos sem ou com poucas perdas na qualidade visual quando o objeto está distante ou pequeno na tela [8]. Para modelos massivos essas técnicas podem ainda assim não serem suficientes se implementadas de forma ingênua. O processamento de visibilidade de objetos pode não escalar quando a ordem for de milhões de objetos, a criação de simplificações de malhas de polígonos e estruturas espaciais para aceleração no processamento da cena como BVH (Bounding Volume Hierarchy), *octrees* e BSP *trees* podem gerar um consumo de memória proibitivo para esses modelos. Em nossa abordagem propomos utilizar geometria procedimental com *Shape grammars* para endereçar problemas de escalabilidade e eficiência no processamento de modelos CAD massivos.

Modelagem procedimental tem sido utilizada para criar cenas massivas onde padrões específicos são bem definidos. Padrões de repetição foram utilizados para representar prédios [9] e florestas [10], enquanto padrões de crescimento recursivo foram utilizados em topologia de plantas e fractais [11]. A modelagem procedimental reduz consideravelmente o trabalho manual em domínios com padrões bem definidos. A geração de cenas 3D é feita pela especificação de regras, comumente parametrizáveis, que permitem que o usuário crie modelos rapidamente até que obtenha o visual desejado da cena. Outra vantagem obtida é a redução do consumo de memória, dado que normalmente as regras para expressar um padrão tendem a ser mais compactas que a representação explícita da geometria.

Nesse contexto, *Shape grammars* têm se popularizado como uma técnica de modelagem procedimental dado sua simplicidade e poder de expressão. Entretanto, ainda há desafios para serem utilizados em variados domínios como modelos CAD. Tipicamente, modelagem procedimental gera cenas aleatórias parametrizadas, enquanto modelos CAD devem representar construções do mundo real da forma mais fidedigna possível. Entretanto, podemos observar que modelos CAD apresentam padrões de geometrias e repetições. Essas evidências motivam nossa pesquisa para utilizar *Shape grammars* nesse domínio de forma abrangente, algo que até então não havia sido explorado.

1.1 Objetivos

Essa tese de doutorado propõe a utilização de modelagem procedimental no domínio de modelos CAD massivos. O trabalho tem maior ênfase em resolver problemas de escalabilidade que existem em modelos de plantas industriais. Em outras palavras, estamos interessados em oferecer soluções que

auxiliem na implementação de sistemas de visualização escaláveis de modelos CAD. Modelagem procedimental explora padrões e repetições para gerar cenas 3D, optamos por escolher *Shape grammars*, dado que já possui evidências de serem aplicadas em renderização em tempo real [10, 12] e são expressivas o suficiente para representação de modelos de plantas industriais [13].

Embora nosso foco seja em oferecer soluções para implementação de sistemas de visualização de modelos CAD, temos como contribuição secundária a modelagem de modelos massivos com *Shape grammars*. Estamos propondo a aplicação de uma estrutura de dados flexível que pode ser utilizada de maneira uniforme desde o formato do arquivo até a renderização. Normalmente, existem diferentes formatos e diferentes abordagens para implementar engines de renderização que podem ser inerentes a uma representação (não necessariamente igual ao formato de entrada) ou são genéricas como em engine de jogos. Logo, existe uma constante mudança de contexto em como uma cena 3D é descrita dificultando a implementação de uma renderização direta e eficiente.

Nesta tese destacamos as seguintes contribuições:

- **Gramática para plantas industriais**. Nesse trabalho propomos o MCAD *Shape grammar*, uma gramática especializada em descrever modelos CAD massivos. Dentre os primeiros desafios, destacamos que *Shape grammars* têm sido mais utilizadas para modelagem procedimental de ambientes urbanos, principalmente prédios e fachadas para criar cidades sintéticas virtuais. Os modelos CAD que utilizamos nesse trabalho são da indústria de óleo e gás e seguem rigorosamente regras de um projeto de engenharia em contraste com demais trabalhos que utilizam modelagem procedimental para criar modelos aleatórios. Portanto, nossa gramática é expressiva suficiente para representar qualquer modelo CAD sem perda de informação e sem sobrecarregar a representação, ou seja, sem aumentar complexidade de modelagem ou consumo de memória. Até então, não encontramos nenhum outro trabalho que tenha utilizado *Shape grammar* nesse domínio, o que enfatiza a originalidade dessa pesquisa.
- **Compressão de memória**. Utilizando nossa gramática, conseguimos comprimir a representação de modelos CAD massivos mesmo em tempo de execução. Essa compressão se deve ao fato de que tanto *Shape grammars*, quanto modelagem procedimental geralmente identificam padrões e repetições que um domínio possui e definem regras compactas que geram as mesmas configurações de objetos que uma representação explícita e mais custosa. Dessa forma, a representação promove reuso de regras e parametrização que por sua vez podem ser derivados pela própria semântica da modelagem.

– **Conversão e Otimização de gramática**. Apesar de propormos uma gramática para descrição de modelos massivos, nos preocupamos em converter modelos preexistentes e avaliar nossas propostas em cima dos mesmos. Implementamos uma conversão de modelos que recebe um arquivo em formato CAD comercial e convertemos para nossa gramática de tal forma que possa utilizar as características da geração procedimental. Dentro de otimização de modelos mostramos heurísticas de como melhor explorar a geração de modelos dada uma representação arbitrária de modelos CAD.

– **Engine de MCAD *Shape grammar***. Aliado às contribuições anteriores propomos algoritmos e uma engine que utiliza como estrutura de dados básica um MCAD *Shape grammar*. Nessa engine exploramos a otimização na renderização de objetos primitivos que são comuns em cenas de modelos CAD massivos. Além disso, para aumentar a escalabilidade da engine, propomos um mecanismo de geração de cena *on-the-fly* conforme a visão atual do observador. Nesse esquema, geramos objetos derivando e interpretando regras somente quando necessário, reduzindo assim o processamento realizado pelo sistema. Na engine que propomos, implementamos balanceamento de tarefas entre CPU e GPU, onde no lado da CPU a gramática do modelo é processada em *multithread* gerando os objetos para serem desenhados de forma eficiente no lado da GPU.

Em todas as análises e avaliação deste trabalho utilizamos modelos reais desenhados para projetos de engenharia industrial.

1.2 Organização do trabalho

No Capítulo 2 é apresentada a fundamentação teórica necessária para melhor entendimento desta tese. São apresentados conceitos do domínio de modelos CAD massivos e modelagem procedimental com foco principal em *Shape grammars*. Também, são apresentados os trabalhos relacionados de suas respectivas áreas.

O Capítulo 3 apresenta nossa proposta de gramática para representação de modelos CAD massivo. Este capítulo contém a análise que levou a sua definição, além de exemplos de modelagem que exercitam sua validade para modelagem de modelos CAD.

O Capítulo 4 apresenta uma engine que utiliza o MCAD *Shape grammar* como representação de modelos. Este capítulo contém os algoritmos desenvolvidos para renderização eficiente dos modelos gerados por nossa gramática.

O Capítulo 5 descreve o processo de conversão de modelos pré-existentes em MCAD *Shape grammar*. Enfatizamos também o pré-processamento dos modelos para criar gramáticas que vão gerar cenas eficientes em nossa engine proposta.

O Capítulo 6 apresenta uma avaliação ampla do MCAD *Shape grammar* na criação de sistemas de visualização de modelos CAD massivo. Em seguida faz uma comparação com o estado da arte no domínio modelos CAD Massivos. Por fim, discute os resultados e limitações baseado na implementação desenvolvida nesta pesquisa.

Por fim, o Capítulo 7 apresenta as conclusões obtidas nesta tese bem como os trabalhos futuros que esta propõe.

O Apêndice A contém as equações paramétricas utilizadas para desenhar as primitivas básicas do MCAD *Shape grammar*.

2
Fundamentação teórica

2.1
Modelos CAD massivos

Esta seção faz uma introdução a modelos CAD massivos, destacando características do domínio, problemas e soluções tradicionais da literatura e indústria.

2.1.1
Motivação

Modelos CAD possuem ampla aplicação na engenharia para construção, planejamentos e simulações em plantas industriais. Esse domínio tende a gerar cenas massivas, ou seja, haverá uma alta quantidade de informação para ser visualizada. A Figura 2.1 mostra uma planta industrial de uma refinaria utilizada na indústria de óleo e gás. Podemos ver que o modelo possui um nível de granularidade fino de detalhes (Figura 2.1c) e ao mesmo tempo é denso, ou seja, muitas geometrias presentes em um curto espaço (Figura 2.1b). A quantidade de objetos se deve, além da complexidade das plantas industriais, à forma em que as estruturas são modeladas, geralmente macro objetos são formados pela composição de micro objetos, por exemplo um tanque pode ser modelado pela combinação de um cilindro e semiesferas (vide Figura 2.2).

Em modelos CAD massivos destacamos alguns requisitos básicos que devem ser cumpridos no intuito de se obter uma navegação interativa:

- **Escalabilidade da estrutura dos objetos**: um dos principais gargalos da renderização do modelo está no volume de dados das geometrias que o modelo contém. O custo de desenhar pode estar dividido tanto na quantidade de objetos para serem processados antes da renderização quanto no volume no processamento da geometria em si. Usualmente a geometria é uma malha de polígonos ou uma representação compacta que será transformada em malha no momento da renderização;
- **Compactação de memória**: apesar do problema do consumo de memória estar sendo minimizado com o avanço da tecnologia, pode ser ainda um problema relevante dependendo da modelagem do projeto e

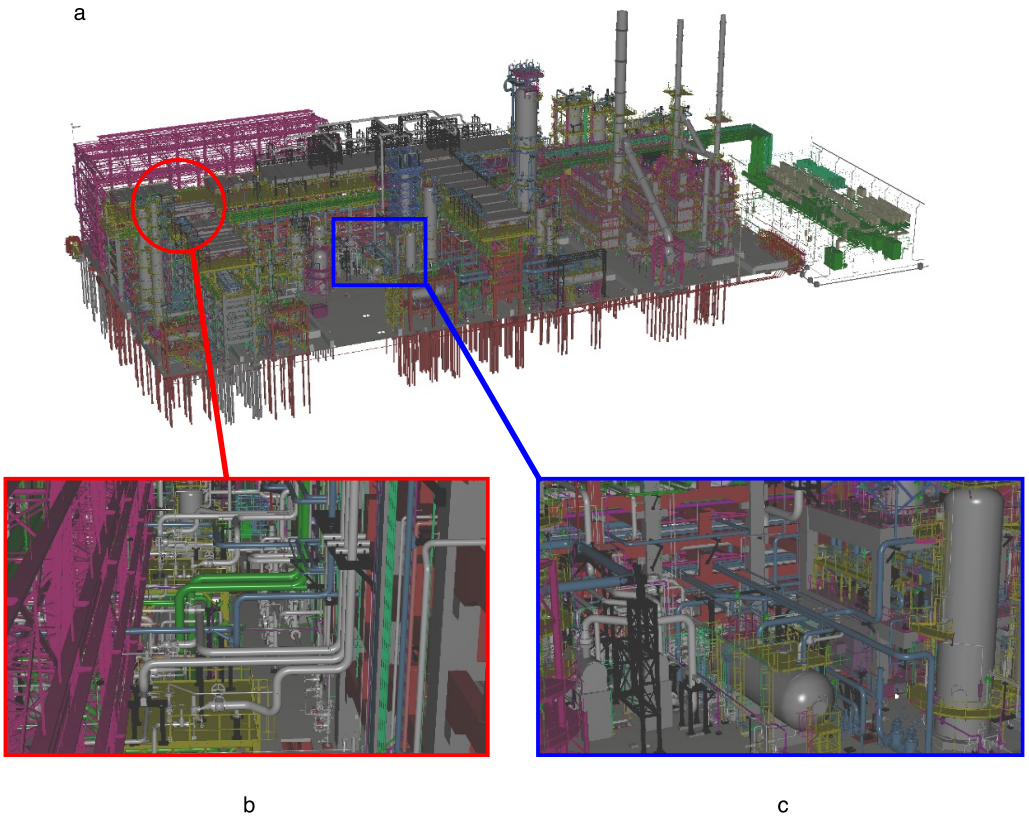

Figura 2.1: Imagem de um modelo CAD de uma refinaria. O modelo contém 929K objetos e 134M triângulos, renderizado a 11 FPS utilizando API de instanciação da GPU em um computador *desktop* com processador Intel Core i7 2.93 GHz Quad Core, 6GB de RAM e placa gráfica Geforce GTX 770. a. Visão geral da planta industrial; b. visão interna das tubulações; c. visão ampliada de um setor.

da plataforma em que o sistema é executado. As malhas de polígonos podem gerar um alto volume de dados, que por sua vez pode causar uma sobrecarga no sistema quando não há memória suficiente para armazenar o modelo completo impactando na interatividade da cena. Essa sobrecarga pode ser originada pela própria estratégia da *engine* com esquemas *out-of-core* ou da paginação do sistema operacional ou do driver da GPU. Deve-se também considerar o consumo de memória de estruturas espaciais extras que vão acelerar o processamento do modelo. Além do mais, o destino final dos dados antes de serem renderizados é a memória de vídeo da placa gráfica que normalmente é mais limitada quando comparada a memória principal do sistema.

Softwares comerciais dividem a complexidade dos modelos de várias formas: pré-processando os dados para otimizar a navegação do modelo, identificando e agrupando tipos de objetos comuns seja para diminuir consumo de memória ou melhorar a renderização, restringindo a qualidade visual da

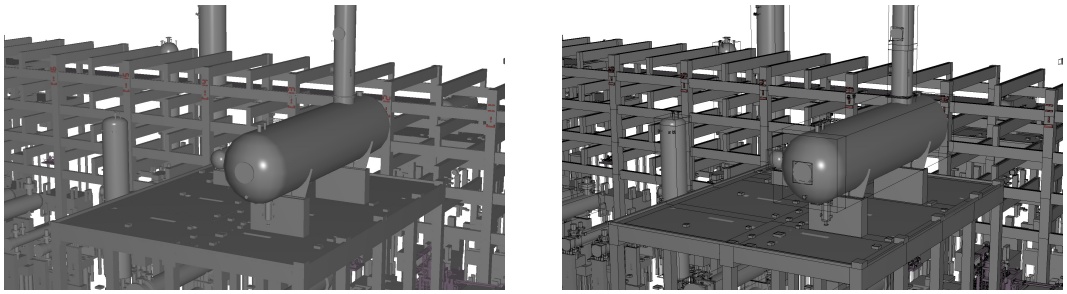

Figura 2.2: Cena de uma modelo CAD. A esquerda a renderização original, a direita a renderização onde cada objeto é desenhado com sua caixa envolvente. Note que embora as estruturas aparentem ser uma única superfície uniforme, na verdade são formadas por um conjunto de objetos menores.

imagem para maior fluidez da cena, dentre outros. O usuário, por sua vez, navega no modelo em subdivisões menores de tal forma que diminua a carga no sistema para que fique compatível com suas configurações de *hardware*, o que pode comprometer a produtividade no seu trabalho.

Além disso, para que esses modelos tenham utilidade efetiva em um projeto industrial, pode ser necessário que sejam navegáveis em cenas com moderado dinamismo. O BIM é uma metodologia de construção que utiliza modelos 3D na coordenação de projetos [1], consequentemente variados casos de usos podem surgir que podem exigir maior riqueza na renderização desses modelos. Dentre as aplicações destacamos planejamento [14, 15, 16], detecção de colisão [3, 17], simulações [4, 18] e etc. Esses casos de uso podem ser refletidos na *engine* em mudança de cor, visibilidade, transparência, texturização e movimentação de objetos. Logo, os requisitos do sistema vão além da renderização de cenas estáticas.

Atualmente muitos problemas em visualização de modelos massivos são minimizados com o avanço da tecnologia do *hardware*, entretanto a complexidade de modelos também avança com a tecnologia atual. O "surgimento"[1] de novas plataformas alvo como capacetes de realidade virtual (RV), realidade aumentada (RA) e *tablets*, oferecem novas formas de interação em ambientes virtuais. Entretanto, a portabilidade de alguns desses dispositivos, *design* e economia de energia vão ser fatores que vão limitar os recursos disponíveis. Os óculos de RV, mesmo usando computadores de mesa para renderização de cenas, exigem requisitos mais difíceis de cumprir: a cena tem que renderizada pelo menos a 90 FPS para gerar imagens estéreas a 45 FPS e não causar des-

[1]Aqui "surgimento" se a refere a recente consolidação comercial. Boa parte dessas tecnologias já existiam há mais de uma década, porém ainda estavam incipientes e experimentais para serem utilizadas em casos de usos mais complexos. Hoje o usuário final já tem acesso a esses dispositivos e empresas têm criado produtos destinados a essas plataformas.

conforto aos usuários do sistema [19]. Logo, existe a necessidade de revisitar as soluções em software de modelos CAD massivos quando for necessário cumprir esses requisitos contemporâneos.

A evolução constante das placas gráficas não só aprimorou o poder de processamento e aumentou a memória como também disponibilizou novas API's que minimizam os gargalos recorrentes das aplicações. Geralmente, o *overhead* de transferência de dados entre CPU e GPU é um dos principais gargalos, que é contornável utilizando instanciação, *tesselation shaders* e programação genérica em GPU que surgiram como extensões a nível de *sofware* (no *driver*) nas placas gráficas ao longo dos anos. Recentemente, o *vulkan*[2] surgiu como uma nova especificação de API de GPU que visa ser eficiente e multiplaforma, inclusive podendo ser disponibilizada em dispositivos com baixo processamento computacional. Essas soluções motivaram a criar soluções mais elegantes no desenvolvimento de sistemas de visualização.

As seções seguintes descrevem um conjunto de soluções clássicas da literatura e utilizadas por softwares comerciais para implementação de visualizadores 3D em tempo real. Não é o objetivo deste documento fazer um apanhado extenso sobre o tópico, caso interesse o leitor, há sugestão de literatura em [21, 22].

2.1.2 Determinação de visibilidade

O cálculo de visibilidade de objetos é um dos requisitos mais básico de qualquer *engine*. A premissa básica é calcular de forma menos custosa se um objeto está visível na cena para então ser enviado ao pipeline de renderização. Considerando comumente que um objeto é representado por uma malha de triângulos, a primeira abordagem é definir uma estrutura espacial mais simples, que consuma menos memória que a do objeto, cujo cálculo de visibilidade vai ser mais rápido que desenhar a geometria do objeto.

Volumes envolventes são os tipos de estruturas mais comuns utilizados como simplificações de malha de polígonos. A decisão do tipo de volume exige um estudo de balanceamento entre precisão do objeto, custo no cálculo de visibilidade e construção da estrutura. Dentre os mais comuns temos AABB (Caixa envolvente alinhada com o eixo), OBB (Caixa envolvente orientada com o eixo) e esfera. A AABB é a mais comumente utilizada em renderização em tempo real de forma genérica. Os algoritmos clássicos de visibilidade consideram AABB como estrutura base, além de ser rápida e simples de construir mesmo para cenas dinâmicas.

[2] [20] `https://www.khronos.org/vulkan/`

A Figura 2.3 ilustra alguns cenários possíveis para determinação de visibilidade de objetos. O algoritmo mais comum e praticamente obrigatório de ser implementado em uma engine é o *view-frustum culling* ou *frustum culling*. O *frustum culling* consiste em utilizar o volume da visão da câmera para testar se objetos estão inclusos para então enviar a renderização. O *occlusion culling* é um algoritmo mais sofisticado, que consiste em determinar se objetos estão realmente visíveis mesmo estando dentro do *frustum* da câmera ou seja, não obstruído por outros objetos. Essa técnica necessita de uma implementação mais complexa que o *frustum culling*; uma abordagem mais simples é ter informação a priori do modelo e armazenar "dicas" que podem ser utilizadas para determinar objetos oclusos de forma eficiente. O *occlusion culling* possui uma boa eficácia em ambiente de interiores numa abordagem chamada de *portal culling* [23] que divide a cena em células e quando um portal está visível, o *frustum* é subdividido em setores definidos por sua intersecção com as células, diminuindo então o espaço de busca do *frustum culling*. Por fim, *detail culling* descarta objetos que não vão ou pouco contribuirão na renderização de um quadro. Em geral vai descartar objetos muito pequenos ou que estão muito distantes da câmera. Essa abordagem de descarte tende a não ser exata e conservativa como as anteriores, sua implementação consiste em estimar a projeção aproximada do objeto na tela de forma rápida e estimar a contribuição na imagem final [24].

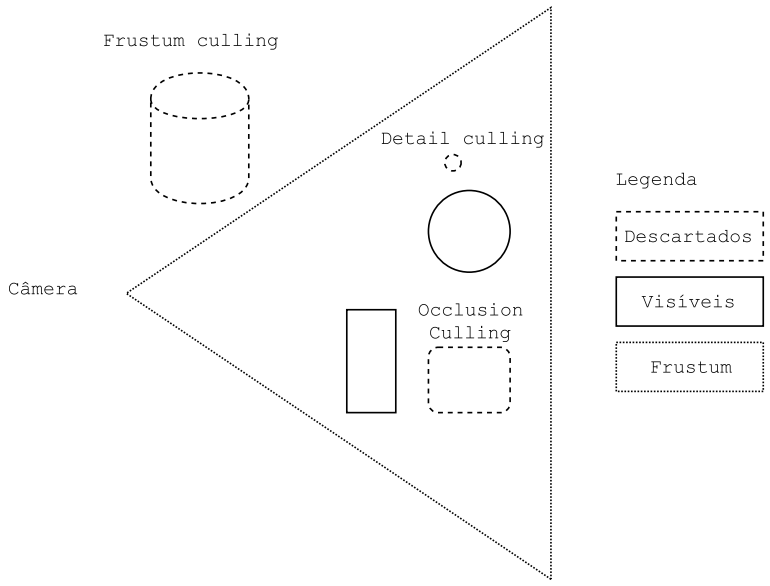

Figura 2.3: Esquema com situações possíveis de determinar visibilidades de objetos em uma cena.

2.1.2.1 *Frustum culling*

Esta seção descobre o algoritmo de *frustum culling* para testes com caixas envolventes (AABB e OBB). Essa implementação considera algumas otimizações comparado ao método direto e geométrico. Uma vez que uma das entradas do método é a matriz *view-projection*, deve-se primeiro extrair os seus planos do *frustum*. Um método simples e facilmente implementável é a extração de planos baseado em espaço de *clip*. Dado um plano definido por (Pn_x, Pn_y, Pn_z, Pw), a Equação 2-1 calcula os planos do *frustum* para uma matriz *view-projection* VP_n, onde n é o número da coluna da matriz, detalhes da derivação em [21].

$$\begin{aligned} P_{esquerdo} &= \quad VP_1 + VP_4 \\ P_{direito} &= -\ VP_1 + VP_4 \\ P_{inferior} &= \quad VP_2 + VP_4 \\ P_{superior} &= -\ VP_2 + VP_4 \\ P_{frente} &= \quad VP_3 + VP_4 \\ P_{fundo} &= -\ VP_3 + VP_4 \end{aligned} \tag{2-1}$$

Então, o próximo passo é calcular se pelo menos um vértice da caixa envolvente está dentro do volume, testando se estes estão no lado positivo dos planos que compõem o *frustum* (considerando que a normal é direcionada para o interior do *frustum*). Uma forma de otimizar, realizando menos teste, é calcular os *p-vertex* e *n-vertex*. Dado um dos planos, o *p-vertex* é o vértice mais distante deste, e o *n-vertex* o seu oposto. Se o *p-vertex* está do lado negativo do plano, a caixa pode ser definida como completamente fora do *frustum*, caso contrário checando apenas o *n-vertex* é possível determinar se o cubo está completamente acima ou em intersecção com o plano. Para uma AABB com seu mínimo (min_x, min_y, min_z) e máximo (max_x, max_y, max_z), uma normal do plano (Pn_x, Pn_y, Pn_z), o cálculo do seu *p-vertex* (Vp_x, Vp_y, Vp_z) é dado por:

$$\begin{aligned} Vp_x &= \begin{cases} max_x, \ se\ Pn_x \geq 0 \\ min_x, \ senão \end{cases} \\ Vp_y &= \begin{cases} max_y, \ se\ Pn_y \geq 0 \\ min_y, \ senão \end{cases} \\ Vp_z &= \begin{cases} max_z, \ se\ Pn_z \geq 0 \\ min_z, \ senão \end{cases} \end{aligned} \tag{2-2}$$

o *n-vertex* (Vn_x, Vn_y, Vn_z) por sua vez, o oposto é dado por:

$$Vn_x = \begin{cases} min_x, \ se\ Pn_x \geq 0 \\ max_x, \ senão \end{cases}$$
$$Vp_y = \begin{cases} min_y, \ se\ Pn_y \geq 0 \\ max_y, \ senão \end{cases} \quad (2\text{-}3)$$
$$Vp_z = \begin{cases} min_z, \ se\ Pn_z \geq 0 \\ max_z, \ senão \end{cases}$$

caso a caixa seja uma OBB, a normal do plano deve ser transformada para o espaço da caixa utilizando seus eixos Ba_x, Ba_y e Ba_z, a nova normal Nb é dada por:

$$Nb = (Ba_x \cdot Pn, Ba_y \cdot Pn, Ba_z \cdot Pn) \quad (2\text{-}4)$$

Por fim, o pseudoalgoritmo a seguir determina se a caixa está dentro, fora ou em interseção com o *frustum* (valores $DENTRO$, $FORA$ e $INSTERSECÇÃO$ respectivamente) recebendo como parâmetros o *p-vertex*, o *n-vertex*, e F como conjunto de planos no espaço de *clip* que formam o *frustum* no formato (Pn, P_w):

Algoritmo 1 Frustum culling

```
função FrustumCulling(F, Vp, Vn)
    resultado ← DENTRO
    para todos P ∈ F faça
        se Vp · Pn < −P_w então retorne FORA
        senão se Vn · Pn < −P_w então
            resultado ← INTERSECÇÃO
        fim se
    fim para retorne resultado
fim função
```

2.1.3 Hierarquias espaciais

Hierarquias espaciais ou estruturas espaciais são estruturas de dados utilizadas para aceleração no processamento de objetos geométricos. Dentre elas destacam-se BVH (*Bounding Volume Hierarchy*), BSP (*Binary Space Partitioning*) *trees* e *octrees*. A BVH aninha volumes envolventes, garantindo que sempre que nós filhos estejam contidos espacialmente pelos seus respectivos nós pai. As BVH's possuem uma fácil e rápida construção mesmo para cenas dinâmicas, e são uma das otimizações mais simples de implementar

em um sistema de visualização. As BSP *trees* divide o espaço em planos recursivamente, nessa partição é possível organizar os objetos de forma que sejam acessados ordenados, enquanto nas BVH's o acesso a objetos é aleatório. A *octree* é um tipo de hierarquia espacial onde cada nó não folha possui 8 filhos, cujas caixas envolventes são octantes da caixa envolvente de seus respectivos pais. As *octrees* podem ser utilizadas para fazer consultas como BSP-*trees*, porém tendem a ser mais compactas dado que sua organização é regular. Na prática essas estruturas vão criar hierarquias de objetos diferenciando-se apenas pela forma de como organizam os objetos no espaço. A Figura 2.4 ilustra os tipos de estruturas simplificado no espaço 2D, onde a *octree* é a apresentada como *quadtree* que é sua análoga para espaços 2D.

Utilizando estruturas espaciais, a complexidade do cálculo de *frustum culling* pode ser dividida reduzindo processamento desnecessários. Considere de forma genérica uma árvore de objetos que utiliza qualquer uma dessas estruturas de espaciais. Iniciando uma travessia pelo nó raiz, é feito um teste checando se o espaço definido que representa um dos nós filhos imediatos está visível ou não, caso não esteja visível, logo é possível descartar toda a hierarquia adiante. A decisão de qual estrutura utilizar vai depender da topologia do modelo, custo de construção e atualização no caso de cenas dinâmicas e consumo de memória.

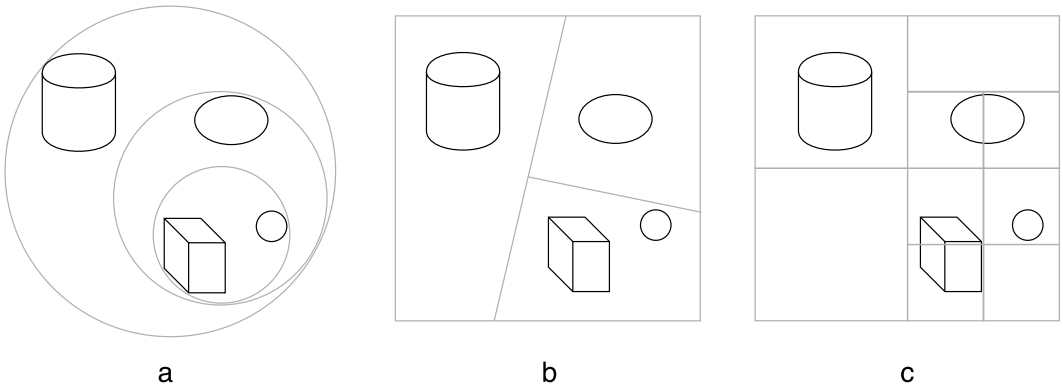

Figura 2.4: Exemplo de hierarquias espaciais. Da esquerda para direita a. BVH com círculos envolventes b. BSP *tree* c. *Quadtree* (equivalente da *octree* em 2D). Note que a BVH e a BSP *tree* no exemplo geram a mesma hierarquia, diferenciando somente pelo tipo de consulta.

Estruturas espaciais combinadas com *frustum culling* são soluções padrões para visualizadores 3D. No domínio de CAD massivo, estas soluções também estarão presentes e resolvem o problema de escalabilidade até uma certa magnitude de modelos. O descarte de ramificações da hierarquia, entretanto não dá benefícios quando a cena é muito densa, como quando o modelo

está visto inteiro na cena como na Figura 2.1a. O *Occlusion culling* por sua vez, atua num problema mal condicionado que é determinar oclusões de objetos, e não há uma solução bem definida e eficiente para casos genéricos.

2.1.4 Level-of-detail

O *Level-of-detail*, mais referenciado como LOD, é uma técnica que consiste em alternar a renderização de objetos por representações mais simplificadas quando este tem menor contribuição na imagem final. As abordagens da técnica podem variar conforme as características do domínio da cena que podem ser otimizadas para melhorar a performance do sistema e a qualidade visual da renderização.

De forma geral existem duas abordagens de LOD: *discreto* e *dinâmico*. No LOD discreto, antes do objeto ser desenhado é calculado sua contribuição na imagem e o resultado deve classificar o nível que o objeto se encontra baseado em algum critério, como distância ou tamanho em tela, para então selecionar sua respectiva versão (vide Figura 2.5). As simplificações devem ser carregadas pelo sistema previamente que podem ter origem de modelagem direta ou utilizando algoritmos de simplificação de malha do objeto quando está com máximo nível de detalhe. O efeito colateral dessa técnica chama-se *popping*, quando a diferença entre as re presentações é muito discrepante e a transição de malhas é realizada de maneira abrupta. Uma forma de minimizar o defeito é utilizando *alpha blend*, que renderiza as duas versões ao mesmo tempo e a que está sendo substituída vai perdendo a opacidade gradativamente até que fique somente a nova versão do objeto.

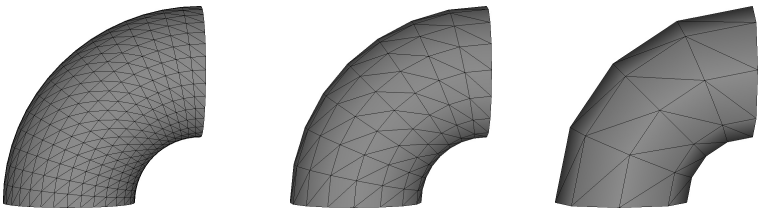

Figura 2.5: LOD's de um toro. LOD 0 - 1024 triângulos, LOD 1 - 256 triângulos, LOD 2 - 64 triângulos.

O LOD dinâmico ou contínuo consiste em realizar uma transição entre versão de malhas de polígonos de maneira mais suave que a discreta. Uma possível implementação é através de *progressive meshes* [25] que realiza um processo chamado de *geomorph* na transição entre versões de geometrias. Resumidamente, o modelo contém uma fila de arestas colapsáveis, que são acessadas

progressivamente para substituir dois vértices por um durante a simplificação. O processo de colapsar é reversível, chamado de divisão de vértices, que é aplicado quando for necessário melhorar a qualidade da geometria até que retorne a sua configuração original.

Para selecionar que LOD do objeto que deve ser desenhado é necessário definir um critério. Esse critério pode variar conforme o sistema, a seleção pode ser baseada em distância, tamanho da projeção em tela, erro em imagem, importância semântica, etc. Um dos mais utilizados e que produz um resultado razoável é a estimativa do tamanho da projeção na tela, que geralmente é estimado baseado no volume envolvente do objeto.

Uma abordagem simples é utilizando esferas envolventes. Dado uma esfera com raio r e centro em c, a câmera posicionada em v, observando na direção d, o raio quadrado projetado na tela p é dado por:

$$p = \frac{nr}{d \cdot (c - v)} \tag{2-5}$$

onde n é o parâmetro *near* da matriz de projeção da câmera. Por fim a área da esfera projetada na tela é dada por πr. Com esse método, a área da projeção é inversamente proporcional à distância da câmera e o objeto.

A área projetada de uma caixa envolvente pode ser calculada usando o algoritmo de [26]. Essa técnica é baseada no fato de que só é possível visualizar no máximo três faces da caixa e consequentemente no máximo 6 vértices (vide Figura 2.6), e que podem ser determinadas num cálculo rápido baseado na visão da câmera. Então, de acordo com a visualização são enumeradas as possíveis situações em uma tabela, onde em cada entrada há os vértices da caixa que formam o fecho convexo naquela situação. Uma vez obtido o fecho convexo, calcula-se a projeção dos vértices na tela e em seguida a área do polígono projetado.

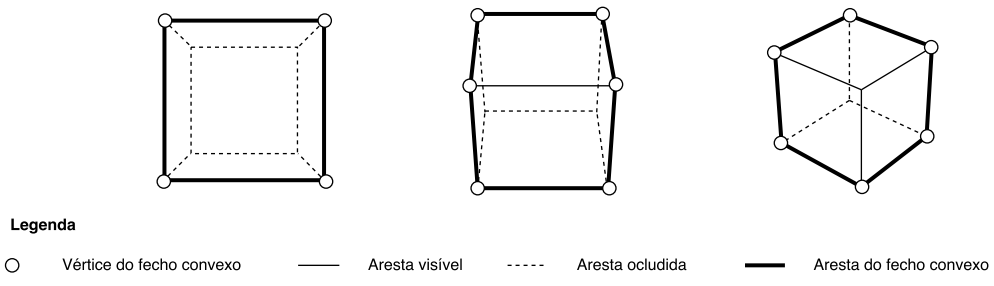

Figura 2.6: Os três casos possíveis de projeção de cubo na tela.

A implementação de LOD em modelos massivos deve ser utilizada com cautela. A quantidade de objetos que um modelo contém pode gerar um consumo de memória proibitivo, quando utilizado LOD discreto. Uma estratégia

simples é aplicar em um conjunto de objetos que são recorrentes como cilindros, toros, semiesferas, e etc. No caso de LOD contínuo, a implementação precisa ser escalável, dado que pode haver milhares de objetos que precisa ser aplicado LOD ao mesmo tempo.

2.1.5 Heurísticas

Em muitos casos, a complexidade de um modelo pode ser tão alta, ou as configurações de *hardware* serem incipientes, que mesmo utilizando as técnicas mencionadas anteriormente não é possível visualizar com taxas interativas. Logo, softwares comerciais de visualização de modelos CAD massivos costumam utilizar heurísticas para melhorar a navegação na cena. A utilização de heurísticas tem o objetivo de renderizar uma imagem aproximada permitindo que o usuário do sistema tenha resposta de navegação com mais fluidez. Em geral, as heurísticas utilizam informações como resultados de processamento de quadros anteriores, resultados parciais do quadro corrente ou postergando o processamento de partes mais complexas do modelo. O efeito colateral é renderização de imagens com erros decorrente do processamento incompleto da cena. A imagem deve ser corrigida imediatamente nos quadros seguintes, o efeito visual é semelhante ao *popping* do LOD, porém, ao invés de uma alteração de resolução em parte da imagem, objetos inteiros surgem de forma incoerente.

Destacamos dois tipos de heurísticas principais que podem serem utilizadas combinadas:

- **Renderização baseada em coerência espacial**, objetos próximos da câmera estão no *cache* da *engine* (seja em CPU ou na GPU) para serem renderizados. Uma navegação lenta ou guiada pode não gerar nenhum erro de renderização, além de ganhar em eficiência pelo uso da heurística. Porém, quando a câmera se movimenta rapidamente pela cena, pode ocorrer um *cache-miss*, quebrando a coerência espacial.
- **Renderização progressiva**. Nesse caso, temos a situação onde um quadro para ser desenhado mesmo após o cálculo de visibilidade e LOD ser muito custoso para manter um FPS desejado. Por exemplo, quando se tem muitos objetos numa mesma visão por conta de uma modelagem com alta granularidade. Uma possível solução é realizar uma ordenação completa ou parcial dos objetos da cena e renderizar aqueles que mais contribuem para a imagem corrente, como os objetos maiores ou mais próximos da câmera.

2.1.6 Trabalhos relacionados

Wald et al. [27] propuseram um sistema *out-of-core* para visualização com *ray tracing* de modelos massivos. Em seu sistema, o modelo é pré-processado e armazenado em arquivo em formato binário. Os dados do modelo são acessados de forma paginada utilizando chamadas do sistema operacional. Para diminuir a latência, o sistema identifica dados que não estão no cache e substitui por voxels chamados de proxies. Esses proxies funcionam de forma análoga ao LOD, e são usados temporariamente até os dados serem carregados de forma assíncrona. Os autores alcançaram 3-7 FPS no Boeing 777 nas configurações de hardware da época.

Gobbetti e Marton [28] introduziram *far voxels*, que substitui detalhes distantes de um modelo por *voxels* que são objetos que servem como simplificações dessas regiões. O modelo é pré-processado para gerar uma hierarquia onde os nós internos podem ser visualizados pelos *far voxels* e os nós folhas os triângulos originais do modelo. O método atinge taxas interativas e funcionam em modelos arbitrários. Entretanto Soares et al. [29] reportam que em modelos CAD massivos semelhantes aos que utilizamos nesta pesquisa, a técnica produz muitos artefatos quando aplicado em regiões compostas por objetos finos.

No trabalho de Peng et al. [30] é apresentado uma abordagem *out-of-core* para renderização de modelos massivos combinado com algoritmos executados na GPU. A transferência de malhas é reduzida por um esquema de coerência quadro a quadro, dado que malhas da imagem atual possuem uma boa probabilidade de estarem presentes na cena no quadro seguinte. Na GPU é realizado simplificação de malhas para realizar LOD diminuindo *overhead* entre CPU e GPU, e um algoritmo de desfragmentação de dados para melhorar o gerenciamento de memória. Dado que o método depende da coerência entre quadros, um movimento rápido da câmera pode causar travamentos na renderização pela transferência de um grande volume de malhas que não estava disponível na GPU.

O trabalho de Santos e Celes [31] apresenta um método para renderizar modelos massivos utilizando instanciação de objetos repetidos na cena. Inicialmente o modelo é processado para encontrar malhas redundantes utilizando *Shape Matching* baseado em [32], que consiste em obter as transformações afins entre malhas. Utilizando esse método é possível diminuir o consumo de memória significativamente e renderizar o modelo de forma eficiente utilizando API de instanciação de malhas presente em placas de vídeo modernas.

Xue et al. [33] propõem uma renderização de modelos massivos utilizando representação de *voxels* com implementação baseada *out-of-core* em GPU.

A representação do modelo em voxel é utilizando para realizar consultas de oclusão e seleção de nível de detalhe para renderização de objetos. Os *voxels* também são utilizados para gerar sombras no modelo no intuito de aumentar o realismo da cena. Para aumentar a escalabilidade os autores propuseram um método de comprimir os dados que vão ser transferidos da CPU para GPU. Os dados dos vértices são quantizados e as normais são descartadas, ao serem transferidos para a GPU os vértices são decodificados no *vertex shader* e as normais são recalculadas utilizando *geometry shader*. Os autores obtiveram uma boa performance no protótipo implementado para modelos massivos, entretanto processo de *voxelization* e compressão de dados causaram perda na qualidade visual da renderização.

Utilizando MCAD *Shape grammar* convertemos modelos com magnitude comparável ao Boeing 777 utilizado em [30, 33], entretanto pudemos renderizar o modelo com taxas interativas sem precisar de um esquema *out-of-core*. A especialização de algumas primitivas que descrevem maior parte do modelo, e o compartilhamento de malhas redundantes reduz o volume de dados de malha na GPU, tornando necessário somente enviar as matrizes de transformação para posicionar os objetos. Estratégia semelhante utilizada também por [31], que obteve bons resultados para modelos massivos, entretanto os autores não implementaram técnicas para aumentar a escalabilidade do sistema como *frustum culling* e LOD. A utilização de superfícies paramétricas também contribuiu para melhorar a qualidade visual dos objetos, diferente de [31] em que as superfícies são discretizadas estaticamente, de [33] que perde detalhes pela compressão de dados da geometria e de [28, 29] que introduz artefatos na cena usando *far voxels* que não correspondem bem a topologia do modelo.

2.2 *Shape Grammar*

O formalismo *Shape grammar* foi introduzido por G. Stiny em 1980 [34] que primeiro definiu o conceito de *Shape* e em seguida *Shape grammar*. Posteriormente Parish e Müller utilizaram *Shape grammars* para modelagem procedimental de cidades [35], no qual foram introduzidos operadores para modificar uma massa de modelo e gerar diversos tipos de arquitetura. A definição de *Shape grammar* utilizada neste trabalho é baseada no trabalho de Müller et al. [9] que introduziram *CGA Shape*.

Um *Shape grammar* $G = (N, T, \omega, P)$ é formado por um conjunto de símbolos (*strings*) não terminais N e terminais T, um símbolo não terminal ω chamado de axioma tal que $\omega \in N$. Um conjunto P de regras de produção que em sua forma mais básica

$$A \rightarrow B \tag{2-6}$$

onde A é um único símbolo e $A \in N$, chamado predecessor, e B um ou mais símbolos, tal que $B \in N \cup T$.

Dado um *Shape grammar*, para geração de uma cena dividimos em dois processos: *derivação* e *interpretação*. A derivação ou processo de produção consiste em expandir as regras de produção da gramática no intuito de gerar símbolos que representam instruções e objetos enquanto que a interpretação consiste em ler e interpretar os símbolos terminais para geração dos objetos propriamente dita. A Figura 2.7 ilustra os dois processos.

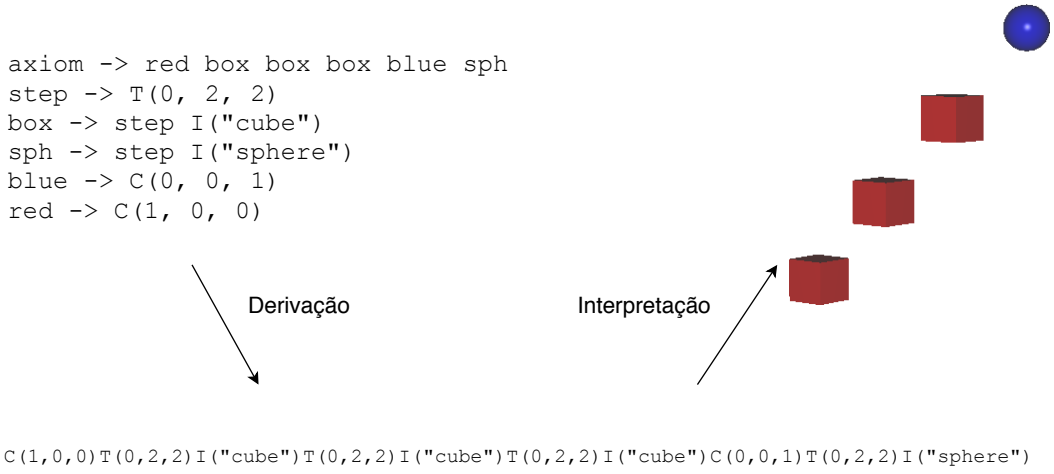

Figura 2.7: Processos de derivação e interpretação de gramática. A derivação expande as regras de produção substituindo predecessores por sucessores até gerar somente símbolos terminais. A interpretação utiliza a saída da derivação para gerar uma cena 3D.

A derivação inicia-se selecionando a regra de produção cujo predecessor é o axioma. A primeira regra de produção deve gerar como sucessores símbolos não terminais e/ou terminais. A produção deve continuar substituindo os símbolos não terminais por seus sucessores até que permaneçam somente símbolos terminais. Os símbolos terminais gerados pela derivação são processados por um interpretador similar ao interpretador tartaruga como é feito em L-system [11]. Todos os símbolos são processados sequencialmente e geram uma instância de objeto ou mudam o estado do interpretador. O estado do interpretador é chamado de *escopo*, que é uma caixa orientada e pode ser transformado por operação como *escala* ($S(x, y, z)$), *translação* ($T(x, y, z)$), e *rotação* ($R(x, y, z)$) combinado com uma *cor* ($C(r, g, b)$) a ser aplicada em um objeto. Os símbolos que representam *instância*($I(\text{“}nome\_do\_objeto\text{”})$) geram objetos aplicando a transformação do escopo atual durante a interpretação.

CGA Shape introduziu operadores específicos para gerar prédios e fachadas em modelagem urbana: *Split* e *Repeat*. O operador *Split* divide o escopo em um ou mais eixos que onde cada parte será substituída por uma regra de produção que deve ser interpretada com a subdivisão do escopo. Possui a seguinte forma:

$$Split(\text{“}XYZ\text{”}, r_1, r_2, ..., r_n)\{p_1, p_2, ..., p_n\} \tag{2-7}$$

os eixos no primeiro parâmetro podem ser um, todos os eixos ou combinação dentre estes, os valores r_n são as razões que dividem cada parte do escopo que por padrão são absolutos, quando for necessário defini-las como relativo basta sufixar com a letra r. Os símbolos p_n são interpretados com o escopo dividido definido pelos parâmetros anteriores respectivos. O operador *Repeat* tem um comportamento semelhante ao do *Split*, entretanto possui apenas um parâmetro que é a quantidade de vezes (*count*) que um escopo será dividido e um único símbolo (*rule*) que será repetido. Possui a forma:

$$Repeat(\text{“}XYZ\text{”}, count)\{rule\} \tag{2-8}$$

esses operadores subdividem o escopo em escopo menores e são úteis para expressar padrões de divisão e repetição.

A seguir, um exemplo de *Shape grammar* com as operações básicas. A regra com o predecessor *axiom* é a primeira a ser avaliada, em seguida os seus sucessores. As regras de produção A, B, C, D, E e F ilustram operações de escopo e operações de divisão, as regras *cube*, *cylinder* e *sphere* encapsulam regras de instâncias e atribuição de cor. A Figura 2.8 mostra a saída gerada pela derivação e interpretação do *Shape grammar* exemplo.

```
axiom -> A B C D E F
A -> cube
B -> T(2,0,0) cylinder
C -> T(2,0,0) R(45,45,0) cylinder
D -> T(2,0,0) S(1.5,1.5,1.5) cylinder
E -> T(2.5,0,0) Split("Z",0.25,0.25,0.25,0.25)
     {cube cylinder cube cylinder}
F -> T(2.5,0,0) Repeat("XYZ", 27){ sphere }
cube -> C(1, 0, 0)I("cube")
cylinder -> C(0, 1, 0)I("cylinder")
sphere -> C(0, 0, 1)I("sphere")
```

Adicionalmente aos operadores de escopo, existem mais dois operadores: *push* ([) e *pop* (]). Utilizando essas duas operações é possível indicar na interpretação que haverá uma ramificação independente. A operação de *push*

salvará o estado do escopo atual em uma pilha, enquanto a operação de *pop* restaurará para o escopo do topo da pilha, removendo-o. Essas operações são semelhantes as utilizadas em L-system, especialmente para modelagem procedimental de plantas que necessitam de ramificação na geração de sua topologia.

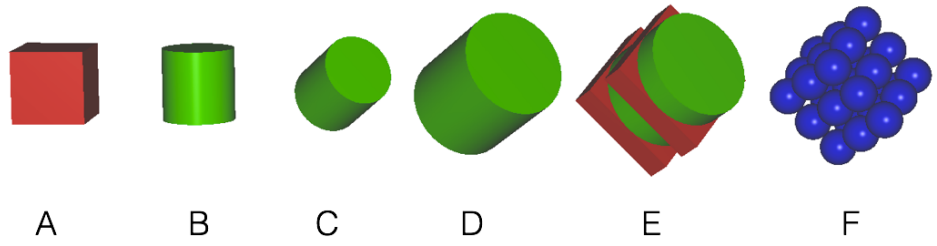

Figura 2.8: Renderização de um CGA *Shape* exemplo, ilustrando todas as operações.

As regras de produção podem ter variações que enriquecem a expressividade da gramática, estas podem ser *parametrizadas* e/ou *condicionais*. Ampliando a forma na Equação 2-6 temos:

$$A(p_1, p_2, ...p_n)[cond] \rightarrow B \tag{2-9}$$

no qual p_n são os parâmetros da regra de produção e *cond* a condição que deve ser satisfeita para que a regra de produção gere seus sucessores durante a derivação. Considere o *Shape grammar* a seguir:

```
axiom -> P(0)
P(x)[x=0] -> I("cube") T(1, 0, 0) P(x+1)
P(x)[x=1] -> I("cylinder") T(1, 0, 0) P(x+1)
P(x)[x<10] -> I("sphere") T(1, 0, 0) P(x+1)
```

utilizando regras condicionais podemos ter várias regras de produção com o mesmo predecessor, quando uma regra de produção deriva um predecessor com múltiplas regras de produção, a escolha é definida pela condição satisfeita conforme o estado da derivação. No exemplo da gramática anterior, a regra de produção com predecessor P é derivada de forma recursiva, mas a derivação é controlada por uma variável passada como parâmetro e condições que checam essa variável em cada possibilidade de derivação. O resultado da geração desta gramática é mostrado na Figura 2.9.

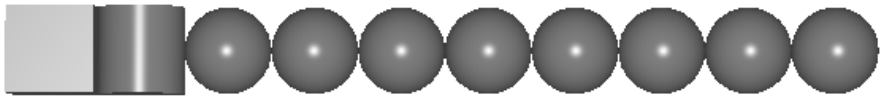

Figura 2.9: Renderização de uma gramática recursiva que utiliza regras paramétricas e condicionais.

2.2.1 Trabalhos relacionados

Modelagem procedimental tem sido utilizada em diversos domínios para reduzir o trabalho manual humano e obter cenas em ambientes urbanos [36, 35, 37, 9, 38]. A maioria desses trabalhos utilizam *Shape grammar* como base para explorar padrões de repetição.

Utilizando *Shape grammars* e *L-systems* como descrição de modelos alguns trabalhos exploraram a geração de cena diretamente na GPU [39, 10, 12, 40]. Essas abordagens obtêm alta escalabilidade uma vez que as regras de geração de cenas são mais compactas que a geometria explícita, assim a sobrecarga de transferência ente CPU e GPU é significativamente reduzida.

Krecklau et al. propôs a linguagem G^2 (*Generalized Grammar*) como uma ferramenta de modelagem procedimental em diferentes domínios [41]. A linguagem tem o poder de expressar prédios, como feito por *Shape grammars*, e plantas, que por sua vez é feito por L-systems. O poder de expressão foi dado pela aplicação de *free-form deformation* em símbolos não terminais. Os autores reportaram que a linguagem pode ser usada em aplicações em tempo real, porém em sua atual implementação não é adequada para ser utilizada em cenas massivas.

Os trabalhos citados obtêm resultados relevantes que tornam a abordagem de modelagem procedimental atraente para ser aplicados em modelos industriais massivos, porém as soluções são específicas e nem sempre podem ser utilizadas diretamente. Esses trabalhos, em geral, apresentam técnicas que geram cenas aleatórias, que para seu propósito atendem de forma satisfatória. Por outro lado, um modelo CAD não pode ser gerado aleatoriamente, todas as estruturas devem ser projetadas e validadas antes de serem utilizadas para tarefas como planejamento. Nesse trabalho, utilizamos as características de gerar cenas procedimentalmente de modelos massivos, porém com desenhos corretos e aplicáveis em projetos de engenharia.

3
MCAD *Shape Grammars*

Este capítulo descreve nossa proposta MCAD *Shape grammar*, um *Shape grammar* especializado para o domínio de modelos CAD massivos industriais. Nossa gramática foi baseado na definição na de Müller et al. [9]. No Apêndice B contém a forma estendida Backus-Naur para o MCAD *Shape grammar*.

3.1
Definição da gramática

Na Figura 3.1 é possível ver diferentes visões de uma mesma cena CAD de uma planta industrial. A imagem inferior contém a renderização da cena com suas cores originais do modelo, nas duas imagens superiores podemos ver a distribuição de tipos de objetos em cores diferentes. Logo se percebe que existe um conjunto de tipos de objetos que são comuns em modelos CAD que atendem a representação de uma planta industrial.

Nossa proposta em criar o formato dessa gramática se inicia ao observar essa evidência. Especializamos primitivas básicas na definição da gramática que são recorrentes em modelos massivos de plantas industriais. Essas primitivas representam em média 70% do modelo, então uma vez que esses objetos são representativos e incorporados na gramática é possível reduzir o consumo de memória, dado que não é necessário que cada modelo redefina a malha desses objetos e também otimizar a renderização dessas primitivas como descrito no Capítulo 4.

Nas seções seguintes vamos introduzir algumas extensões nos operadores de escopo e especialização de primitivas do operador instância.

3.1.1
Escopo

As operações de escopo geram a transformação de um objeto no modelo. Nós definimos as operações de escopo por dois tipos: *relativa* e *absoluta*. As operações absolutas recebem parâmetros absolutos, no qual o resultado não terá influência direta da configuração atual do escopo. Por exemplo, pode-se mover um escopo para um ponto específico independente da sua posição corrente através de uma translação absoluta. As operações relativas, por outro

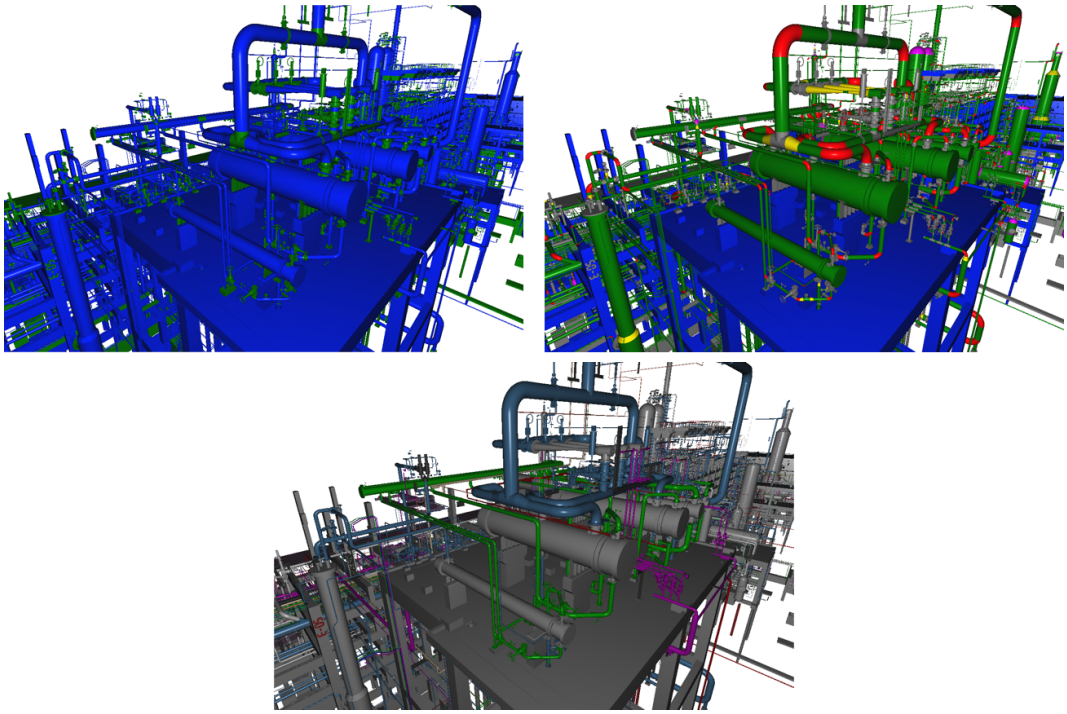

Figura 3.1: Visões diferentes de uma mesma cena em modelo CAD industrial. À esquerda acima, mostra-se a distribuição entre objetos paramétricos (azul) e malhas genéricas (verde). À direita acima, distribuição entre diferentes tipos de objetos: caixas (azul), cilindros (verde), cones (amarelo), semiesferas (magenta), toros (vermelho) e malhas (cinza). Abaixo, cena com as cores originais dos objetos no modelo.

lado, sofrem influência do escopo atual quando aplicadas. A seguir o sumário das operações de escopo:

- $T(x, y, z)$ **Translação relativa**: translada o escopo por (x, y, z);
- $M(x, y, z)$ **Translação absoluta**: move o escopo para (x, y, z);
- $R(x, y, z)$ **Rotação relativa**: rotaciona o escopo por (x, y, z) em ângulos de Euler;
- $G(x, y, z)$ **Rotação absoluta**: atribui ao escopo a rotação a (x, y, z) em ângulos de Euler;
- $S(x, y, z)$ **Escala relativa**: escala o escopo por (x, y, z);
- $E(x, y, z)$ **Escala absoluta**: atribui o escopo a escala (x, y, z).

As operações de escopo absolutas são úteis para transformar um objeto dado que em alguns casos não é possível tirar vantagens da modelagem procedimental. Por exemplo, quando é necessário instanciar um objeto que não tem coerência com o escopo que o pressupõe. Logo, as operações absolutas vão garantir que os objetos sejam posicionados corretamente conforme a especificação do projeto.

3.1.2 Instância

O operador (I(“*nome_do_objeto*”)) é usado para gerar um objeto de nome “*nome_do_objeto*”. Tipicamente, o nome da instância referencia uma geometria para ser renderizada com a transformação do escopo atual. Como foi dito na Seção 3.1, dado que existem um conjunto de objetos que são comuns para representar uma cena, incorporamos essas primitivas diretamente no conjunto de símbolos terminais da gramática, ou seja, esses símbolos estarão sempre presentes no vocabulário de qualquer MCAD *Shape grammar*. Definimos então esses objetos como *primitivas especializadas*, que estão enumeradas a seguir:

- **Cubo (*cube*)**;
- **Cilindro (*cylinder*)**;
- **Cone (*cone*)**;
- **Esfera (*sphere*)**;
- **Semiesfera (*dish*)**;
- **Toro (*torus*)**.

As primitivas cubo, cilindro, esfera e semiesfera são o que chamamos de *primitivas normalizadas*. A Figura 3.2 mostra a renderização de caixas e cilindros com diferentes escalas. Podemos perceber que ao alterar a escala podemos obter objetos com semântica distintas embora utilize a mesma geometria como base. Os parâmetros desses objetos são alterados implicitamente pela escala, por exemplo, o cilindro tem seu raio definido em X e Y e altura em Z, na Figura 3.2a, a primitiva *cylinder* poderia ser usada para representar um objeto em forma de disco ou parte de uma tubulação como na Figura 3.2b . Na Figura 3.2c por sua vez, a primitiva *cube* é usado como uma caixa, enquanto na Figura 3.2d essa primitiva é usada para criar um plano, que poderia ser um piso de algum edifício, por exemplo.

Em contraste as primitivas normalizadas, as primitivas cones e toros possuem parâmetros explícitos. Os cones têm raio inferior, raio superior, deslocamento em x e deslocamento em y, apresentado na seguinte forma:

$$I(\text{“}cone\ raio\_inferior\ raio\_superior\ deslocamento\_x\ deslocamento\_y\text{”}) \tag{3-1}$$

A Figura 3.3 mostra a renderização de diferentes cones pela alteração do parâmetro passado na operação de instância. Na Figura 3.3c, a combinação de parâmetros gerou um cilindro dado que os raios são iguais, porém com um deslocamento no eixo X.

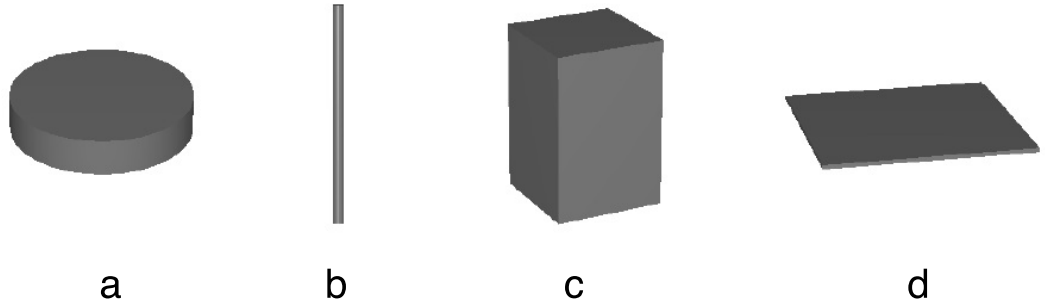

Figura 3.2: Primitivas normalizadas com diferentes escalas aplicadas. Da esquerda para direita: a. Cilindro com escala (0.5, 0.5, 0.1) b. Cilindro com escala (0.1, 0.1, 1) c. Cubo com escala (1, 1, 1.5) d. Cubo com escala (0.5, 0.5, 0.01)

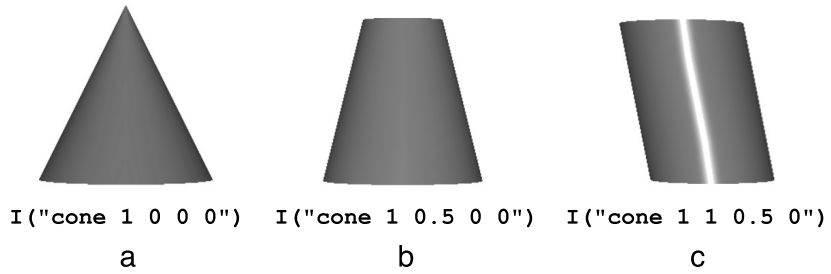

Figura 3.3: Cones gerados utilizando a instância *cone* com diferentes parâmetros. Da esquerda para direita: a. Cone com raio inferior 1 e raio superior 0, sem deslocamento. b. Cone raio inferior 1 e raio superior 0.5, sem deslocamento. c. Cone com raio superior e inferior 1 e deslocamento em x de 0.5.

Os toros possuem ângulo de varredura, raio interior e raio exterior, esses objetos podem ser gerados invocando o operador instância como na forma a seguir:

$$I(\text{“}torus\ angulo\ raio\_interior\ raio\_exterior\text{”}) \tag{3-2}$$

a Figura 3.4 mostra exemplos de toros gerados utilizando esse operador de instância.

Os parâmetros explícitos do cone e toros são normalizados antes de serem desenhados para que fiquem relativos ao escopo, além de centralizados no centro do mesmo. Essa operação é necessária para que o escopo seja sempre a caixa envolvente do objeto, logo também vai exigir que seja aplicada uma escala que vai gerar os objetos com os parâmetros corretos. Nas Figuras 3.3 e 3.4 a maioria dos parâmetros estão normalizados, com exceção da Figura 3.4c que possui raios arbitrários, mas que preservam a proporção desejada para gerar o toro correspondente, nesse caso o equivalente seria atribuir para os raios superior e inferior 0.1 e 1 respectivamente.

Ao utilizar o operador de instância para gerar malhas genéricas, o com-

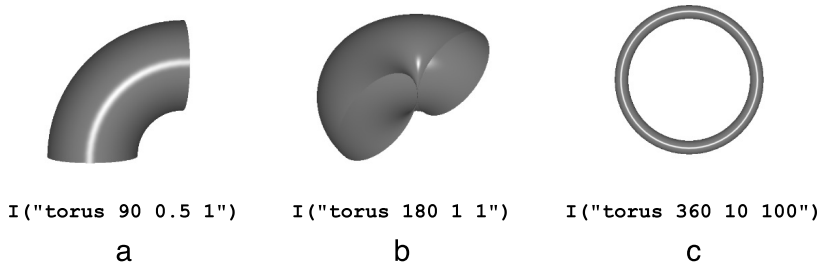

Figura 3.4: Toros gerados utilizando a primitiva *torus*. Da esquerda para direita: a. Toro com ângulo de 90^0 raio interior 0.5 e exterior 1. b. Toro com ângulo de 180^0 raio interior e exterior 1. c. Toro com ângulo de 360^0 raio interior 10 e exterior 100.

portamento da MCAD *Shape grammar* é semelhante às abordagens anteriores de *Shape grammars*. Entretanto, adicionamos uma restrição na configuração das malhas que serão instanciadas. Todas as malhas devem ser normalizadas, ou seja, todos os vértices devem estar sempre contidos no escopo em que serão instanciados. Dessa forma, o escopo sempre será a caixa envolvente orientada de qualquer objeto a ser gerado na cena. A normalização poderá deformar a malha, porém a proporção pode ser recuperada facilmente por uma operação de escala extraída da caixa envolvente da malha original. A Figura 3.5 mostra uma malha deformada pela normalização e a mesma na proporção do objeto no modelo após uma utilizar a operação de escala.

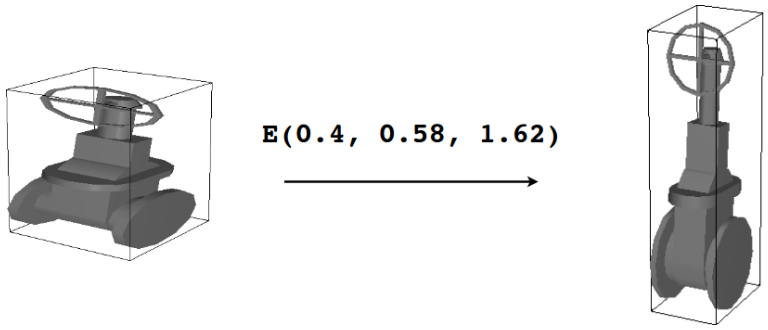

Figura 3.5: Malhas de polígonos genéricas instanciadas com escopos diferentes. À esquerda, a malha instanciada com o escopo inicial deformada com a normalização; à direita a geometria ajustada na proporção original do objeto após a aplicação de uma operação de escala absoluta.

Mais detalhes sobre os parâmetros implícitos e explícitos das primitivas especializadas podem ser consultados no Apêndice A.

3.1.3 Cor

Em modelos CAD 3D, a cor sólida de um objeto, em geral, é suficiente para representar os objetos. Em diversos projetos, a cor pode ter um uso

semântico e não ser necessariamente para ser semelhante ao objeto real. No *MCAD Shape grammar* é definida cor semelhante ao *CGA Shape*, podendo ser alterado por $C(r, g, b)$ ou $C(c)$ onde r (vermelho), g (verde), b (azul) são os componentes da cor ou c como inteiro de 32 bits em hexadecimal.

3.2 Exemplos de modelagem

A seguir mostramos exemplos de modelagem em MCAD *Shape grammar* de padrões presentes em modelos CAD industriais. Esses exemplos exercitam a expressividade de nossa gramática e mostram como geração procedimental pode ser utilizada para gerar sistematicamente tipos de estruturas e criar *templates* reutilizáveis em modelos industriais.

Tanque reservatório. A regra de produção *tank* define um *template* de tanque reservatório que terá como comprimento o valor atribuído no parâmetro *length*. O *length* é usado na operação de escala absoluta que definirá o escopo inicial da geração, a operação escala igual os componentes X e Y e no componente Z é passado o parâmetro dessa regra de produção. A operação de *Split* divide o escopo inicial no eixo Z em três partes, $front$, *body* e *back*. Note que o parâmetro de razão respectivo de *body* possui o sufixo *"r"* que por sua vez significa ser relativo as demais proporções, nesse caso essa parte da divisão do escopo tomará tudo que restou das partes absolutas. Os demais parâmetros da operação *Split* são absolutos, no intuito de fixar o tamanho dessas partes no equipamento. A Figura 3.6 mostra exemplos de tanques reservatórios gerados procedimentalmente pela gramática com diferentes parâmetros de *length*.

```
tank(length) ->
E(2.3,2.3,length+1)
Split("Z",0.5,1r,0.5){front body back}

body -> I("cylinder")

front ->
R(0,180,0)I("dish")
front_connector
[support][bottom_connector]

back ->
I("dish")
[support] bottom_connector

front_connector ->
```

```
T(0,0,0.25)E(0.6,0.6,0.25)I("cylinder")
T(0,0,0.125)E(1,1,0.1)I("cylinder")

bottom_connector ->
T(0,1.3,-1.3)R(90,0,0)E(1,1,0.1)I("cylinder")
T(0,0,0.17)E(0.7,0.7,0.25)I("cylinder")

support ->
T(0,0.95,-2.1)E(2.1,0.1,0.8)R(90, 0, 0)
I("support_mesh")
```

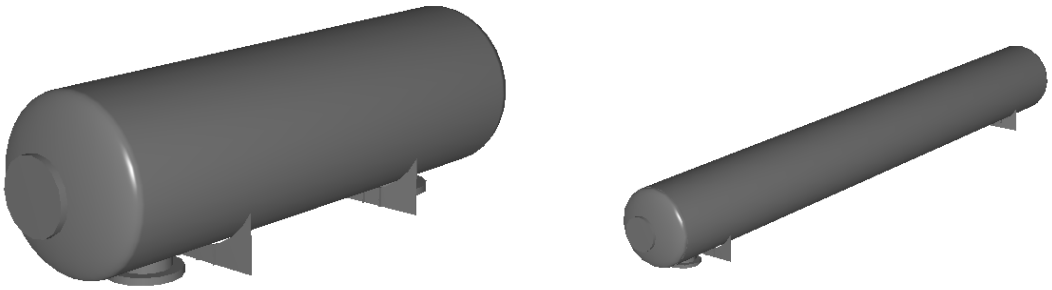

Figura 3.6: Renderização de objetos gerados pela regra de produção *tank*. Na esquerda foi passado como parâmetro $length = 8$ e na direita $length = 20$. Nota: Os dois objetos possuem o mesmo raio de comprimento, porém a diferença de tamanho na imagem é referente a perspectiva na câmera quando renderizado.

Escadas. A próxima gramática possui regras de produção que geram uma escada com corrimãos. A regra de produção *stairs* tem como parâmetro *step_count* que define a quantidade de degraus que a escada gerada deve ter. A regra de produção $lifelines$ gera os corrimãos e recebe como parâmetro o *step_count* e seus sucessores usam esse parâmetro para realizar cálculos que ajustam os objetos que vão formar o corrimão. A regra de produção *step_size* apenas realiza uma escala absoluta que irá definir o tamanho dos degraus da escada. A operação *Repeat*, que nessa forma está sem eixos especificados, irá repetir as regras de produção *step* pelas quantidades de vezes que foi atribuído na variável *step_count*. A cada derivação da regra de produção *step* é instanciada uma caixa e depois aplica-se uma translação relativa que irá preparar o escopo para o próximo degrau a ser instanciado.

```
stairs(step_count) ->
lifelines(step_count)
step_size Repeat("", step_count){step}
```

```
step_size -> E(20, 4, 1)

step -> I("cube")T(0, 4, 4)

lifelines(step_count)->
[T( 10, 0, 0) lifeline(step_count)]
[T(-10, 0, 0) lifeline(step_count)]

lifeline(step_count) ->
E(1, 1, 20)
base_lifeline(step_count)
lateral_lifeline(step_count)

base_lifeline(step_count) ->
[T(0, 0, 10)I("cylinder")]
[T(0, step_count*4, step_count*4+10)
 I("cylinder")]

lateral_lifeline(step_count) ->
S(1, 1, 0.2828*step_count)
T(0, step_count*4*0.5, step_count*4*0.5+20)
[R(-45, 0,0)I("cylinder")]
[T(0, 0, -6)R(-45, 0,0)I("cylinder")]
[T(0, 0, -12)R(-45, 0,0)I("cylinder")]
```

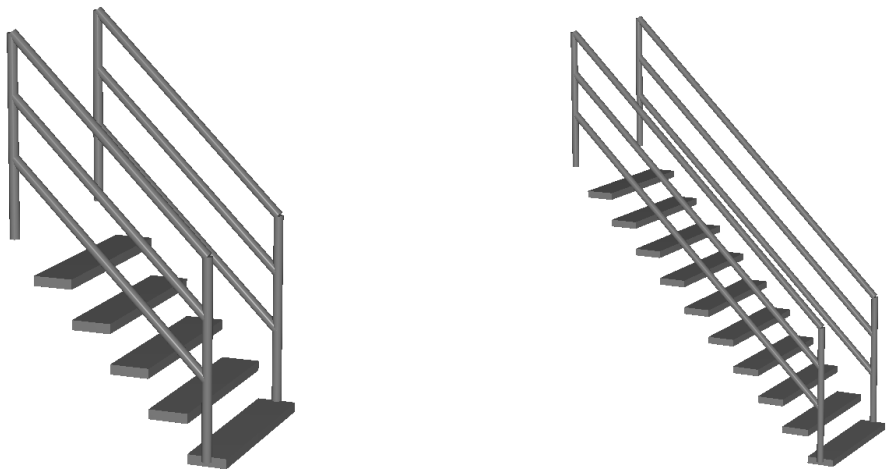

Figura 3.7: Renderização de escadas com a regra de produção *stairs*. A esquerda com $step\_count = 5$. A direita com $step\_count = 10$.

3.2.1 Discussão

Modelos CAD industriais, apesar de tenderem a serem massivos, possuem repetições e padrões de estruturas intrínsecas à natureza desses modelos. Utilizando MCAD *Shape grammar*, sistemas podem expressar templates de objetos ou estruturas de forma parametrizável usando regras de produção paramétricas. Essa parametrização promove a adição de semântica diretamente na modelagem, uma vez que os parâmetros podem incorporar especificação de projetos e restrições na configuração dos objetos em um modelo CAD. Projetos industriais seguem normas de engenharia que definem a configuração dos equipamentos e estruturas, que por sua vez terá reflexo direto nos seus respectivos modelos CAD.

Por exemplo, a regra de produção *tank* produz tanques reservatórios com tamanhos variados atribuindo o parâmetro *length* como visto na Figura 3.6. Pode-se observar que não é somente aplicar uma escala global nos objetos que compõem o equipamento. Nessa regra de produção somente o corpo principal, representado por um cilindro, é escalado enquanto as demais estruturas são preservadas com distâncias e tamanhos definidos pelas demais regras de produção. Em outra abordagem de modelagem, a escala aplicada no cilindro do tanque poderia ser calculada pela capacidade de volume do tanque reservatório necessária no processo que este equipamento seria integrado. No caso da regra de produção *stairs*, a distância entre degraus e seus respectivos tamanhos poderiam ser definidos por uma especificação de segurança que o projeto deve satisfazer, assim como a configuração dos corrimões que devem acompanhar os degraus da escada.

Quanto ao nível de abstração de modelagem, MCAD *Shape grammar* (assim como *Shape grammar* em geral) permite que regras encapsulem outras regras, oferecendo então, que usuários possam escolher entre modelagem de mais baixo nível ou de alto nível. Como visto nas gramáticas apresentadas anteriormente, a modelagem em baixo nível pode não ser tão atrativa à primeira vista. Idealmente, na modelagem de um projeto que utilize MCAD *Shape grammars*, as regras de produção básicas estariam prontas e apresentadas em um catálogo ou biblioteca, então a modelagem consistiria em escolher regras e atribuir os parâmetros necessários para modelagem de uma parte do modelo em alto nível. Por exemplo, a regra de produção *stairs* pode ser usada somente para gerar escadas com quantidades arbitrárias de degraus, porém caso seja necessário alterar a forma que os degraus são representados, basta alterar a regra de produção *step*. A regra de produção *step* instancia uma caixa no escopo gerado pela operação *Repeat* e ao alterar essa regra poderia ser gerado qual-

quer outro tipo de objeto ou até mesmo realizar derivação mais complexa de várias outras regras de produção. Dessa forma, alterando somente uma regra de produção, pode-se alterar todas as instâncias instantaneamente de *template* de uma estrutura do projeto.

Por fim, quando não necessário ou possível utilizar regras procedimentais para gerar objetos, instanciar uma malha arbitrária na cena pode ser feito de forma análoga à regra de produção a seguir:

```
object -> M(10,0,0) E(2,5,1) G(0,0,90) C(1, 0, 0) I("a_mesh")
```

na qual consiste de operadores de escopo absolutos, cor e a instanciação de uma geometria identificada por um nome. Dessa forma, MCAD *Shape grammar* é expressivo suficiente para descrever modelos CAD que representam suas cenas por uma transformação (escala, rotação e translação)[1], cor e malha de objeto e essa representação não é mais custosa do que abordagens tradicionais como grafos de cena [42, 43].

[1]Em nossa abordagem não consideramos outros tipos de transformações, como por exemplo cisalhamento, porém as demais transformações não são utilizadas em modelos industriais explicitamente na transformação.

4
Engine MCAD *Shape grammar*

Este capítulo descreve uma *engine* para visualização de modelos CAD massivos utilizando MCAD *Shape Grammar* como representação. A eficiência do renderizador é dada pelo desenho otimizado das primitivas especializadas descritas no Capítulo 3. A escalabilidade, por sua vez, é dada pela derivação e interpretação *on-the-fly* da gramática, que baseado na visão do observador na cena, seleciona as regras de produção mais adequadas a serem expandidas no intuito de reduzir o custo computacional de processar os objetos da cena, tanto em CPU quanto em GPU. Dessa forma, um sistema baseado em MCAD *Shape grammar* consegue gerar procedimentalmente quais conjuntos de objetos serão enviados a GPU para serem desenhados, reduzindo não só o *overhead* entre CPU e GPU, como também mantendo uma representação compacta de um modelo massivo em tempo de execução.

4.1
Visão Geral

A Figura 4.1 mostra a visão geral em alto nível do fluxo e processamento de dados na *engine* implementada para *MCAD Shape grammar*. Uma linha tracejada separa dois momentos do sistema: carregamento do modelo e navegação no modelo. No carregamento do modelo é feita uma leitura da gramática do sistema do arquivo, seja em formato binário[1], ou através de um *parsing* no formato texto utilizando as convenções apresentadas nos capítulos anteriores.

Após o carregamento da gramática em uma estrutura de dados em memória, é realizada uma travessia[2] por todas as regras de produção buscando pelos operadores instâncias. O formato da gramática não contém informação de malha de polígonos, caso o operador instância referencie um objeto que não faz parte do conjunto de primitivas básicas, o nome do objeto deve ser usado como referência para ser buscado em uma fonte externa. Uma opção é realizar uma leitura de arquivo que tenha o mesmo nome do objeto contendo

[1]Criamos um formato binário do MCAD *Shape grammar* no intuito de acelerar o carregamento de modelos massivos e reduzir o consumo de memória em arquivo, dado que o formato texto é verboso. Os dois formatos são equivalentes e intercambiáveis quando carregados em tempo de execução.

[2]Não confundir com a derivação da gramática, nesse caso visitamos todas as regras de produção sem checar condições e sem realizar repetições dessas regras de produções.

os dados da geometria, usualmente uma malha armazenada em arquivo no formato Wavefront OBJ [44]. As malhas de polígonos são compostas por um vetor de vértices, e um vetor de índices de vértices arranjados de três em três para formar triângulos. Todas as malhas são enviadas à GPU, idealmente, evitando armazenar malhas redundantes. O retorno do *upload* dos dados deve ser o id do *buffer* da memória de vídeo em que foi armazenada a malha que pode ser compartilhada por mais de uma ocorrência do seu respectivo operador instância.

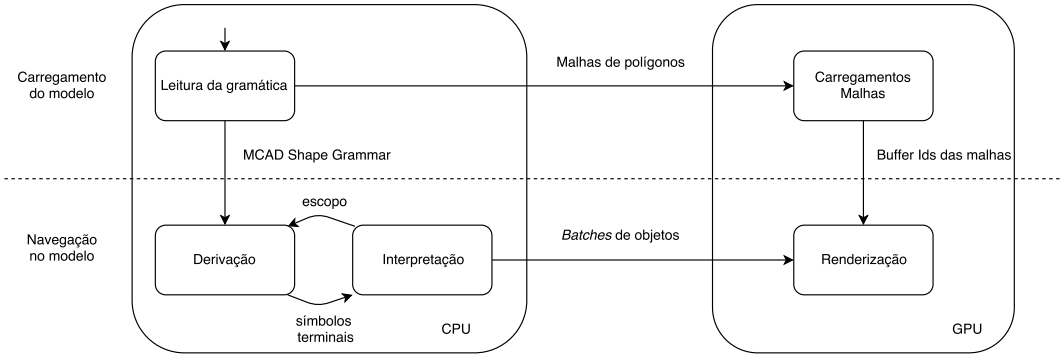

Figura 4.1: Visão geral da engine MCAD *Shape grammar*.

Uma vez que a gramática do modelo é carregada pelo sistema, é possível iniciar a navegação no modelo. A navegação é orientada pela visão da câmera que pode ser modificada tanto por um caminho de câmera predefinido ou por entradas do usuário usando a interface gráfica do sistema. A cada quadro a ser desenhado, a gramática é derivada e reinterpretada gerando os *batches* de objetos a serem desenhados. Em nossa abordagem realizamos derivação e interpretação da estrutura do modelo em MCAD *Shape grammar* simultaneamente em CPU, utilizando regras de produção paramétricas e condicionais onde o principal parâmetro é a visão da câmera (mais detalhes na Seção 4.2). Esses dois processos, quando finalizados, irão produzir os *batches* de objetos que serão enviados à GPU para renderização. No pipeline de renderização foram implementadas otimizações para desenhar objetos comuns do domínio, principalmente as primitivas especializadas (vide Seção 4.4).

4.2 Derivação e Interpretação *on-the-fly*

A derivação e interpretação simultânea processa o MCAD *Shape grammar* em CPU, gerando os *batches* de objetos que serão enviados à GPU para renderização a cada quadro. Enquanto os símbolos não terminais estão gerando seus sucessores (derivação) os símbolos terminais estão gerando suas instâncias

e alterando o escopo imediatamente (interpretação). Usualmente, esses dois processos são realizados separados [39, 11], propomos uma abordagem que os realiza simultaneamente no intuito de ter no contexto da derivação o escopo da interpretação. Nesse cenário, é possível avaliar o escopo durante a expansão de uma regra de produção, ou seja, combinando a caixa envolvente presente no escopo junto a matriz *view-projection* da câmera é possível realizar cálculos de visibilidade procedimentalmente.

A informação de escopo é apresentada como duas variáveis incorporadas no processo de derivação que são: *scope* e *lod*, a serem detalhadas nas Seções 4.2.1 e 4.2.2. Ambas variáveis podem ser utilizadas nas condições de uma regra de produção a serem avaliadas numa derivação. Obviamente, determinar o valor das variáveis causa um custo computacional adicional ou até mesmo causar um comportamento inesperado na renderização do modelo. A utilização dessas variáveis na gramática é discutida na Seção 5.5, que descreve como utilizar esse recurso para construir gramáticas otimizadas para serem renderizadas de forma correta e eficiente.

4.2.1 Variável *scope*

A variável *scope* armazena um valor inteiro que define a visibilidade do escopo atual. A semântica do valor dessa variável é compreendida pela aplicação de operadores lógicos com valores pré-definidos. A seguir a lista de possibilidades de resultados obtidos da aplicação de operadores lógicos com a informação de escopo.

- **scope < 0**, verdadeira se o escopo não estiver visível pelo *frustum* da câmera, ou descartado por detail *culling*;
- **scope $= 0$**, verdadeira se a posição da câmera estiver dentro do escopo;
- **scope $= 1$**, verdadeira se o escopo estiver em intersecção com o *frustum* da câmera;
- **scope > 1**, verdadeira se o escopo estiver completamente contido no *frustum* da câmera.
- **scope $>= 1$**, verdadeira se o escopo estiver visível externamente, seja com intersecção ou não.

À primeira vista, pode-se perceber que com essa variável em uma condição é possível realizar *frustum culling* (em CPU) avaliando uma regra de produção. Essa avaliação pode ser tanto aplicada a um objeto individual como em um conjunto de objetos. De fato, o cálculo da variável *scope* em tempo de

execução é o resultado do teste de *frustum culling* em caixas orientadas como descrito na Seção 2.1.2.

4.2.2 Variável *lod*

A variável *lod* armazena um valor numérico referente ao nível de detalhe que o escopo atual se apresenta para a câmera. Quando há a ocorrência da variável na condição de uma regra de produção é estimado o tamanho da projeção do escopo na tela, conforme descrito na Seção 4.6. A seguir, a descrição de possíveis condições que utilizem a variável *lod*:

- **lod < 0**, verdadeira se a estimativa de tamanho do escopo estiver abaixo do limiar de *detail culling*;
- **lod = 0**, verdadeira se o escopo estiver com tamanho que exija o máximo de detalhe;
- **lod = n, lod > n, lod < n, lod >= n ou lod <= n**, compara *lod* com outro valor definido em *n*.

A variável *lod* permite que na gramática de um modelo possam ser definidas alternativas em regras de produção que são selecionadas conforme o nível de detalhe do escopo. A seguir, um exemplo de uma gramática que utiliza *lod* na condição de regras de produção:

```
object [lod=0] -> I("object_full_mesh")
object [lod=1] -> I("object_coarse_mesh")
object [lod>2] -> I("object_coarsest_mesh")
```

Quando a regra de produção com predecessor *object* é avaliada, em cada uma das possibilidades um objeto será instanciado conforme o LOD mais adequado, similarmente como é feito em LOD discreto tradicionalmente. Entretanto a seleção de LOD, em nossa abordagem, irá ser usada para expandir uma regra de produção, seguindo a abordagem procedimental. Considere a próxima a gramática a seguir:

```
object [lod=0] -> a_complex_production_rule
object [lod=1] -> I("a_mesh")
object [lod>1] -> I("cube")T(1,0,0)I("cylinder")T(0,1,0)I("cube")
a_complex_production_rule -> another_rule ...
another_rule ->  ...
...
```

nesse caso, as regras de produções utilizam LOD para gerar outros tipos de comportamentos. Por exemplo, no caso do $lod > 1$, três objetos são gerados baseados nas primitivas básicas, que podem ser representativos suficientes para a cena, ou até mesmo expressar uma semântica na navegação nesse modelo. Quando $lod = 1$, uma malha arbitrária é renderizada, por fim $lod = 0$ pode encadear diversas regras de produções que vão instanciar objetos para gerar a máxima representação da estrutura criada pela regra de produção com predecessor *object*.

4.2.3 Instanciação

Esta seção descreve como são processados e armazenados os *batches* de objetos gerados pela interpretação do MCAD *Shape grammar* pela invocação do operador instância. Para cada operação de instância, são extraído o escopo e a cor atual resultantes da interpretação que vão formar os atributos do objeto, então existem três cenários possíveis que vão depender do nome do objeto passado como parâmetro:

- **Primitiva normalizada**, todos os parâmetros do objeto estão implícitos no escopo e será desenhada como superfície paramétrica como descrito na Seção 4.4;
- **Primitiva com parâmetros explícitos**, no caso do toro e cone, devem ser extraídos seus parâmetros adicionais e concatenados aos atributos dos objetos;
- **Malha arbitrária**, os atributos são armazenados em *buffers* que estão associados a malhas para *instanciação* da GPU.

Para cada tipo de objeto haverá um *batch* que contém uma referência para um *instance buffer id* onde estão armazenados os atributos de todas as suas instâncias. No caso das malhas arbitrárias, cada geometria única é considerada como um tipo de objeto. A Figura 4.2 ilustra a estrutura do *buffer* de atributos a ser enviado à GPU, que pode variar conforme o tipo de objeto a ser renderizado. As primitivas normalizadas e malhas arbitrárias irão utilizar o leiaute do *instance buffer id'*, no caso das primitivas com parâmetros explícitos irão utilizar o leiaute análogo a *instance buffer id"*. Nessas estruturas de *buffer* é possível renderizar os objetos com baixo *draw calls*, que será igual a quantidade de tipos de objetos a serem desenhados em um determinado quadro. A redução de *draw calls* é primordial para reduzir o *overhead* entre CPU e GPU, que por sua vez impacta no desempenho da *engine*.

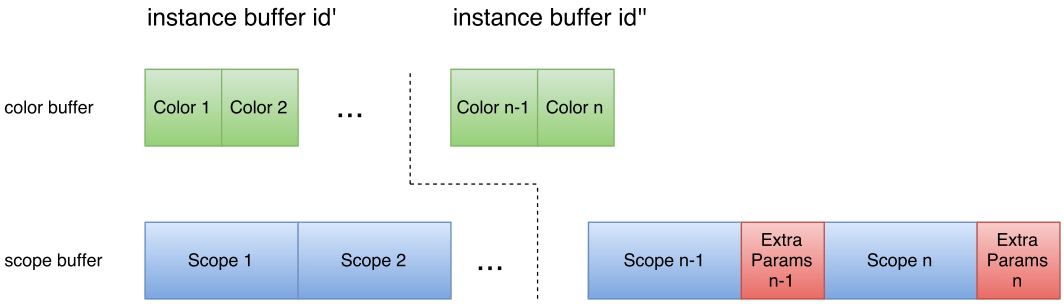

Figura 4.2: Leiaute do *buffer* a ser enviado à GPU para renderizar instâncias de objetos. À direita para primitivas normalizadas e malhas genéricas (*instance buffer id'*), abaixo para primitivas especializadas que necessitam de parâmetros adicionais (*instance buffer id"*).

Os *batches* de objetos são transferidos para os *buffers* da placa gráfica na renderização através de atualização de *buffers* previamente alocados, evitando o custo de alocação de memória que normalmente é muito custoso para ser realizado a cada quadro. Esses *buffers* são alocados na inicialização da *engine* e possuem tamanhos máximos iniciais constantes. O tamanho dos *buffers* pode variar conforme a configuração do sistema, mas idealmente deve ser suficiente para comportar o máximo de objetos visíveis, ou que contribuem significativamente, na renderização de um quadro. Existem dois *buffers* principais: *color buffer* que armazena as cores das instâncias e *scope buffer* que armazena os escopos além dos parâmetros explícitos de algumas geometrias. O *instance buffer id* na verdade é uma abstração para um deslocamento de *bytes* nesses *buffers*. Em nossa abordagem assumimos que realizando cálculos de visibilidades nunca serão renderizados todos os objetos de um modelo massivo. Caso os buffers cheguem no limite pode se considerar três possibilidades de implementação:

- **Alocar mais memória**, nesse caso deve-se gerenciar a memória para que não esgote o recurso da placa gráfica;
- **Liberar memória**, desenhar os *batches* que estão no *buffer* imediatamente e liberar memória para receber os novos, nesse caso a renderização de um quadro vai exigir mais *draw calls* e *uploads* de data que o esperado no cenário ideal.
- **Não renderizar todos os objetos**, sendo que nesse caso pode haver uma reorganização de prioridade que deve preservar os objetos mais significativos na cena, como descrito na Seção 2.1.5.

4.3 Interpretação e derivação *multithread*

O processamento do modelo descrito em MCAD *Shape grammar* é realizado em CPU utilizando múltiplas *threads*. A quantidade de *threads* vai depender da configuração do sistema, que pode ser baseada na quantidade de *cores* disponíveis no processador ou pela quantidade de recurso que se deseja alocar para a aplicação. Entretanto, é esperado que haja no mínimo duas *threads*: uma *thread* para interpretação e derivação e outra para renderização, que chamamos de *updater* e *renderer* respectivamente. A *thread updater* é responsável por gerar os *batches* pelo processamento da gramática baseada na navegação na cena e a *renderer* somente os recebe, faz *upload* para a memória da placa de vídeo e renderiza sem nenhum processamento adicional. Quando há mais *threads* disponíveis, a *updater* pode escalar tarefas a estas e então distribuir o processamento em um esquema de *fila de tarefas*.

O paralelismo das tarefas de renderização e processamento da gramática dá um ganho no desempenho na navegação do modelo. Enquanto a *renderer* está ocupada enviando chamadas à GPU, a updater está processando o próximo quadro como é feito em *engines* modernas. Essa desassociação pode oferecer duas opções de renderização:

- **Renderização síncrona**, a *thread* de *renderer* só renderiza quando o *batch* que recebe é o mais atual;
- **Renderização assíncrona**, a *renderer* quando termina de consumir o *batch* de objetos, inicia o processamento do próximo *batch* independente de ser o mais atual ou não.

Na renderização assíncrona, a *renderer* está desenhando constantemente o que tiver disponível com os parâmetros de câmera atualizados. Então, caso a câmera se movimente muito rápido ou faça rotações bruscas, a imagem renderizada pode não conter todos os objetos, sendo corrigida nos quadros seguintes quando o *updater* terminar de processar a gramática. Embora se caracterize como uma renderização *errada* do quadro, no domínio de modelos massivos esse problema pode ser tolerado quando se abre mão da imagem sempre correta em prol da fluidez na navegação. A taxa de FPS, além da complexidade do modelo, vai depender então principalmente do poder de processamento da placa gráfica.

O desempenho do *updater* vai impactar em ambas as renderizações, seja na fluidez da síncrona, ou na corretude das imagens geradas pela assíncrona. Logo, existe a necessidade de otimizar o processamento da gramática em CPU. Processadores modernos oferecem múltiplos *cores* permitindo processamento

paralelo, logo, uma alternativa é distribuir as tarefas em *threads* que serão escalonadas em unidades diferentes do processador. Chamamos essas *threads* de *workers* que são alocadas na inicialização da *engine* e são acessadas através de um esquema de *pooling*[3].

O primeiro desafio então, se dá em como dividir o algoritmo em processamento paralelo, garantir o compartilhamento de dados entre as *threads*, evitando condições de corrida e *overheads* de sincronização. Como explicado na Seção 2.2 e 4.2, o processamento da gramática, por ser procedimental, possui uma natureza sequencial. Logo, a paralelização do algoritmo não é trivial, a interpretação e derivação simultânea necessitam da informação do escopo atualizada a cada avaliação de regra de produção, que por sua vez vai depender dos símbolos gerados anteriormente. Entretanto, existem as operações de *push* e *pop* que têm a característica de gerar ramificações na geração procedimental, o que vai nos possibilitar paralelizar o processamento da gramática *on-the-fly*. Então, durante a derivação, quando ocorrer uma operação de *push*, os próximos símbolos até a ocorrência do próximo *pop*, junto com o escopo atual são enviados para a fila de tarefas onde podem ser consumidos pelos *workers* realizando a paralelização.

A Figura 4.3 mostra o diagrama de sequência em alto nível que ilustra as interações entre *threads* do sistema na renderização síncrona. A *updater* envia os *batches* (*sendBatches()*) para a *renderer* e em seguida inicia um novo processamento do próximo quadro (*process()*) enquanto a thread de *renderer* está desenhando (*draw()*). Durante o processamento da gramática na *updater*, tarefas paralelizáveis podem ser geradas através da operação de *push* e *pop*, quando isso ocorre a *updater* envia a tarefa para a fila (*work_queue*) (*addTask()*) e esta busca uma *worker* ociosa para escaloná-la. Ao encontrar, a *worker* é ativada (*wake()*) e imediatamente procura por tarefas na fila (*getTask()*), a *thread* inicia a execução que também pode gerar outras tarefas a serem adicionadas na fila. Quando a *updater* termina de processar a ramificação principal da gramática, consulta a fila de tarefas checando se está vazia (*waitToBeEmpty()*). As *workers*, quando finalizam o processamento, consultam e consomem por novas tarefas que estão na fila até que fique vazia. Quando a fila fica vazia (*empty*), a *updater* sincroniza com as *workers* para consolidar os *batches* de objetos do quadro. Eventualmente, a *renderer*, se já estiver terminado de desenhar (*wait()*), vai aguardar a *updater*, que no final da iteração vai preparar os *batches* e então enviá-los para renderização.

[3] *Pooling* consiste em alocar recursos estáticos que são reutilizáveis, uma vez que é mais custoso alocar mais recurso quando necessário. Um gerenciador do *pool* é responsável pelo estado do recurso, se está livre ou ocupado, e retornar os recursos livres quando solicitados.

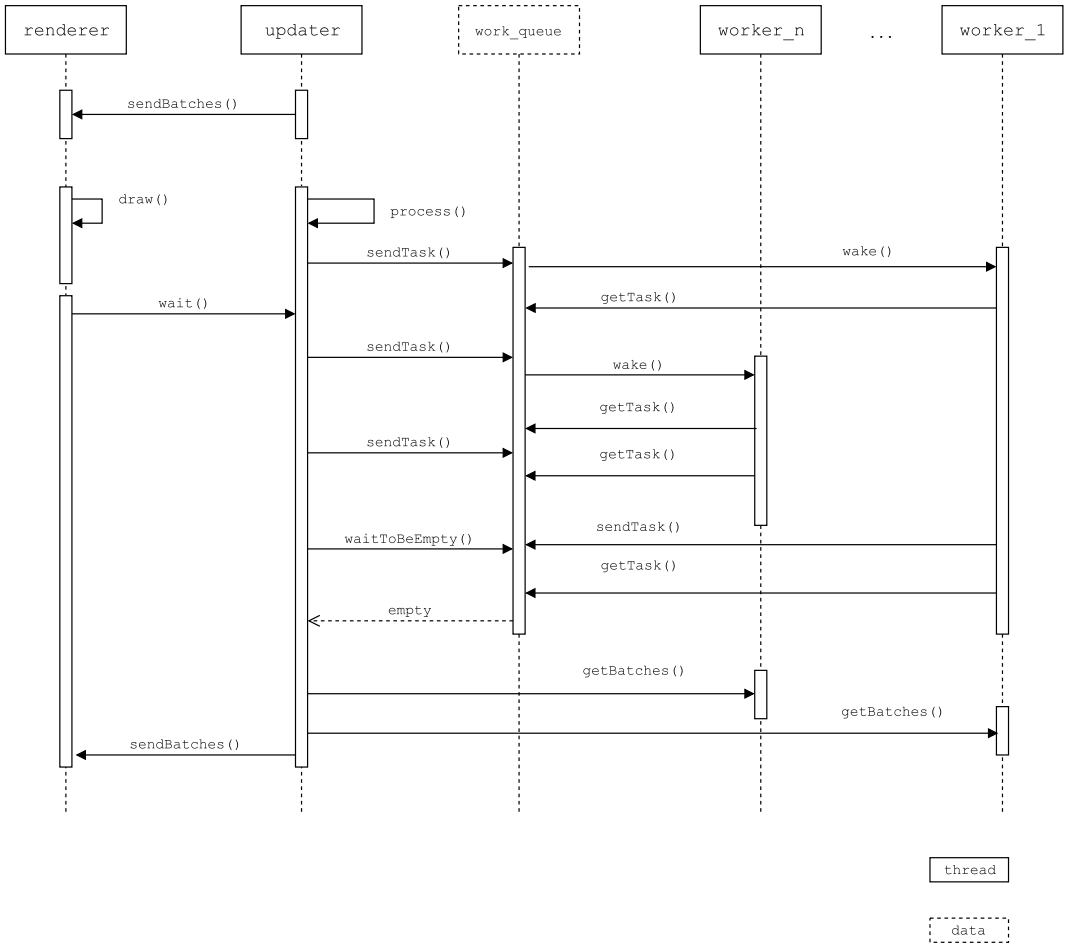

Figura 4.3: Diagrama de sequência ilustrando as interações entre as *threads updater*, *renderer* e *workers* durante a renderização síncrona. Nota: *work_queue* é um objeto, não uma *thread* como a legenda indica.

4.4 Renderização de superfícies paramétricas

As primitivas especializadas do MCAD *Shape grammar* podem ser representadas como *superfícies paramétricas* $\overrightarrow{p} : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ ou seja, uma função 2D que tem como saída um ponto 3D. A representação paramétrica dá características interessantes para as primitivas especializadas como descrição compacta além de permitir gerar um conjunto infinito de objetos diferentes com discretização controlável. O Apêndice A mostra as equações paramétricas das superfícies e a visualização de seus parâmetros explícitos e implícitos.

GPUs modernas oferecem *tessellation shaders* que possibilitam programar na pipeline da placa gráfica a geração de triângulos ou linhas, que por sua vez podem definir a resolução de um objeto de forma eficiente. *Tessellation shaders* foram usados anteriormente para renderização de terrenos com resolução dinâmica em tempo real [45], simulação de cabelo [46] e renderização de

superfícies NURBS [47]. Nós utilizamos esses *shaders* para gerar os triângulos de uma superfície paramétrica diretamente na GPU, controlando a discretização ou até mesmo descartando o objeto baseado no seu respectivo escopo. Resumidamente, geramos uma malha base (2D) cujo vértices servem de parâmetros para serem utilizados em uma equação paramétrica, então "dobramos" a malha base transformando-a na superfície do objeto (3D) a ser desenhado, veja a Figura 4.4.

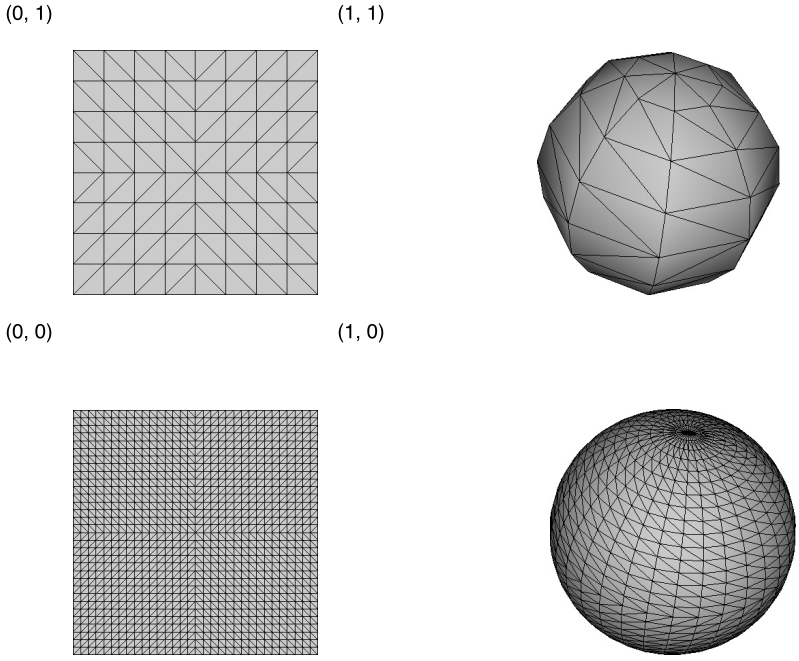

Figura 4.4: Malhas bases geradas pelos *tessellation shaders* com diferentes resoluções e suas respectivas esferas após a aplicação de equações paramétricas. Note as coordenadas que foram utilizadas como parâmetros no cálculo de superfície do objeto.

A Figura 4.5 mostra o pipeline de renderização de uma instância como superfície paramétrica[4]. No *vertex shader* os atributos do objeto são apenas passados para o próximo estágio. No *tessellation control shader* os atributos proveniente do *vertex shader* são processados, onde é realizado LOD e *culling* baseado no escopo (vide Seção 2.1.2 e 4.6). O processamento realizado no *tessellation control shader* define os parâmetros de configuração da tecelagem para serem passados à etapa de *tessellation* que por sua vez é realizada pelo *hardware*. A GPU gera os triângulos da malha base que representam os parâmetros para serem usados de entrada nas equações paramétricas das superfícies. O cálculo da posição dos vértices na superfície é realizado na etapa seguinte pelo *tessellation evaluation shader*. Para cada tipo de primitiva,

[4]Excluímos o cubo, dado que já possui discretização mínima e a representação paramétrica não é trivial. Nesse caso o cubo é armazenado em um buffer para instanciação.

existe um *tessellation evaluation shader* especializado que aplica equações paramétricas da superfície na malha base para gerar o objeto propriamente dito. Por fim, no *fragment shader* é realizado cálculo de iluminação e atribuição de cor como tradicionalmente é feito.

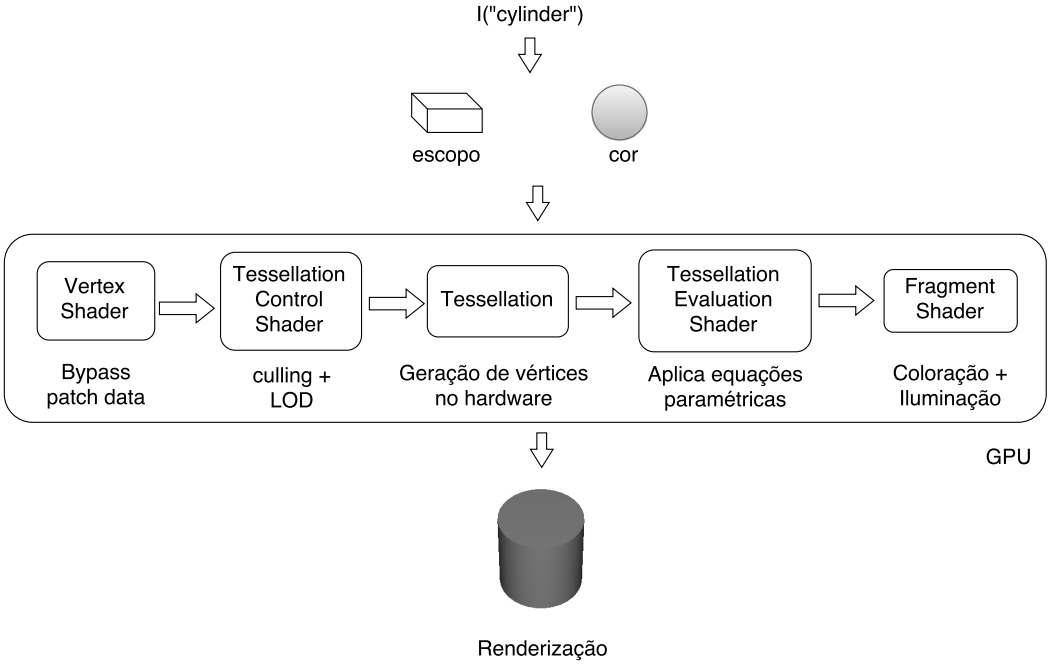

Figura 4.5: Pipeline para renderizar primitivas especializadas. Inicia-se pela chamada do operador instância e obtendo o escopo atual através da interpretação. Os dados são enviados para a GPU onde cada shader especializado irá processar os parâmetros e gerar a superfície do objeto diretamente na placa gráfica.

4.5 Cálculo do *frustum culling*

O escopo, como apresentado anteriormente, é análogo a matriz *model* do pipeline de renderização tradicional, pois define a translação, rotação e escala de um objeto. Entretanto o escopo definido em MCAD *Shape grammar* define que deve ser coincidente com a caixa envolvente do objeto. Isto é, dado uma transformação de um objeto, ao aplicá-la em uma caixa com dimensões unitárias centralizadas na origem, o resultado deve ser a OBB da geometria no espaço do mundo. Isso nem sempre é verdade para a matriz *model*, por exemplo, a malha pode já estar transformada no espaço do mundo e a matriz pode ser identidade, sendo que em CPU é utilizada para definir o valor da variável *scope*, que não necessariamente vai realizar descartes de objetos.

4.6 Cálculo de LOD

Em GPU, se o objeto não foi descartado, o *shader* então deve atribuir o nível de tecelagem conforme a projeção do objeto na tela. Utilizamos a implementação baseada em esferas envolventes para estimar a contribuição do objeto na imagem. Dado que o valor da área projetada é inversamente proporcional à distância da câmera, utilizamos essa métrica para realizar LOD contínuo nas superfícies paramétricas renderizadas em GPU. O valor estimado pondera a resolução da malha base conforme a distância e tamanho do escopo do objeto (vide Figura 4.4). Dado (R_x, R_y) como resolução da malha base na tecelagem do objeto no LOD 0, s um escalar que é o tamanho da projeção ideal do objeto para malha base, a resolução da superfície do objeto para um tamanho de projeção p é dada por:

$$(R'_x, R'_y) = \frac{p}{s}(R_x, R_y) \tag{4-1}$$

Em CPU também optamos por utilizar esferas envolventes para o cálculo da variável *lod*, porém como foi discutido na Seção 4.2.2 o cálculo é utilizado para determinar um valor discreto. O resultado do cálculo também é utilizado para realizar *detail culling* em CPU para diminuir o volume dos *batches* de objetos. Definindo como parâmetro do sistema um tamanho base para ser o LOD 0, os demais LOD's são definidos proporcionalmente a esse valor. A faixa valores são descritas na Tabela 4.1, onde o *lod* é o valor de teste em percentual.

LOD 0	**LOD 1**	**LOD 2**	**LOD 3**
$lod \geq 100\%$	$100\% > lod \geq 50\%$	$50\% > lod \geq 25\%$	$lod < 25\%$

Tabela 4.1: Faixas de valores LOD, *lod* o valor de teste.

4.7 Resumo da configuração do sistema

A Tabela 4.2 mostra o sumário de parâmetros de configuração da *engine* baseado na descrição das seções anteriores.

4.8 Discussão

4.8.1 Qualidade da renderização

As primitivas especializadas representam maior parte do modelo, o que motivou a especialização da renderização desses tipos de objetos. Usando *tes-*

Parâmetro	**Descrição**
Tipo de renderização	Síncrona ou assíncrona
Tamanho do *buffer* de *batches*	Memória de vídeo utilizada para armazenar os atributos dos objetos
Quantidade de *workers*	Quantidade de *threads* extras que consomem e produzem tarefas paralelizáveis durante o processamento da gramática em CPU
Limiar de detail *culling*	Valor de corte para desenhar objetos muito pequenos na tela
Tamanho base do LOD 0	Tamanho mínimo em espaço de tela que define maior resolução de um objeto (ou derivação de forma genérica)

Tabela 4.2: Parâmetros da *engine* utilizados na configuração do sistema.

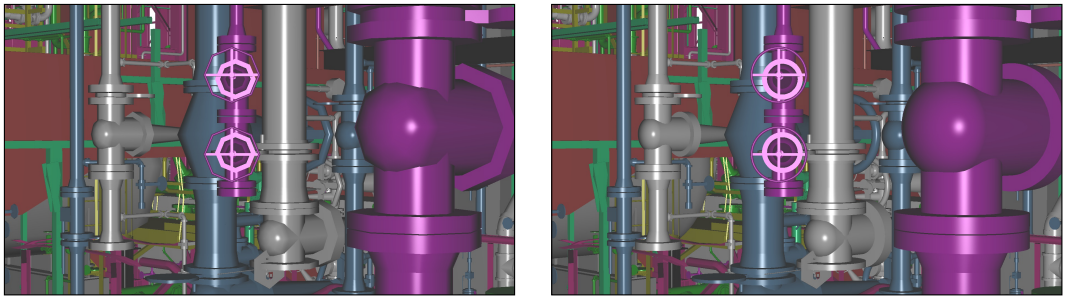

Figura 4.6: A esquerda, cena de um modelo com baixa discretização (malhas base com resolução 16×16). A direita, cena com maior discretização, onde as superfícies ficam mais suaves (malha base com resolução 64×64).

sellation shaders podemos renderizar as primitivas como superfícies paramétricas com discretização controlável realizando LOD contínuo. O benefício direto disso, além da eficiência, é a qualidade visual dos objetos da cena, como mostrado na Figura 4.6, que ilustra o impacto da discretização da malha base na suavização da superfície do objeto. Embora algumas primitivas especializadas sejam mais recorrentes que outras, como é apresentado na Seção 6.2, a composição entre as superfícies suaves vai impactar na qualidade do arranjo de objetos.

4.8.2 Processamento do modelo

A variável *lod* integrada na derivação permite que um sistema de visualização realize a técnica de LOD em um nível acima dos métodos tradicionais.

Dado que o *lod* é utilizado para alternar regras de produção, esta pode derivar qualquer tipo de configuração de objetos, inclusive aninhar outras regras de LOD, em contraste as abordagens tradicionais que trocam o objeto por uma versão simplificada.

A variável *scope* adiciona no processamento da gramática da cena a possibilidade de realizar *frustum culling* em grupos de objetos que são gerados por regras de produção. As regras de produção podem ser utilizadas para criar hierarquias espaciais, onde os sucessores de uma regra de produção são análogos a nós filhos dos predecessores. Logo, a criação da estrutura espacial consiste em configurar o escopo que precede a regra de produção. A configuração da estrutura espacial como BVH ou uma variação da BSP *tree* pode ser feita então de forma procedimental, inclusive utilizando as operações de *Split* e *Repeat* que são especializadas em dividir o escopo.

Sobre *occlusion culling*, embora não tenhamos abordado diretamente nessa implementação da engine no processamento de dados, é possível criar regras de produção que podem alcançar o mesmo resultado. Considere a gramática a seguir:

```
building[scope<0] -> inner_building
building[scope=0] -> outdoor_environment
building[scope>=1] -> outer_building
outer_building -> building_structure outdoor_environment ...
inner_building -> building_structure indoor_environment ...
outdoor_environment -> trees streets ...
indoor_environment -> desks chairs ...
building_structure -> ...
```

esta gramática descreve um modelo que contém pelo menos um prédio (*building*) e seu ambiente externo (*outdoor_environment*) e interno (*outdoor_environment*). A regra de produção *building* tem duas possibilidades de derivação que vão depender do escopo em que é chamada. Se a câmera estiver dentro do escopo (*scope<0*), a derivação prossegue em *inner_building* que por sua vez vai renderizar a estrutura do prédio (*building_structure*) junto aos objetos que fazem parte do seu interior. Se a câmera estiver fora do prédio, com o prédio visível (*scope>=1*) o prédio é renderizado junto ao seu ambiente externo (*outdoor_environment*). Nessas duas circunstâncias houve descartes de objetos semelhante ao comportamento do *occlusion culling*, tanto dos objetos internos do prédio quando a câmera está fora, quanto aos objetos externos, quando dentro do prédio. Por fim, se o *scope=0*, o prédio é descartado junto ao seu interior de forma aninhada, semelhante a abordagem de BVH. Note o

reuso de regras de produção nessa gramática, promovido pela geração procedimental. A escrita dessas regras pode ser baseada na semântica, quando se conhece a topologia do modelo como foi descrito no exemplo anterior, ou pré-processando o modelo, detectando objetos que podem estar obstruídos durante a navegação no modelo.

4.8.3 Gerenciamento de memória da GPU

Uma das configurações da *engine* é o tamanho do *buffer* de *batches* (Tabela 4.2) que armazena os atributos dos objetos para serem desenhados a cada quadro. Por esse *buffer* ser fixo há o risco de no processamento dos objetos da cena, gerar objetos que não cabem no *buffer* alocado. Em nossa implementação, assumimos que o *buffer* é suficiente para desenhar os objetos mais representativos do quadro e ignoramos a possibilidade de não caber. Na Seção 4.2.3 apresentamos soluções para contornar o problema da falta de memória. Um momento oportuno de estimar o tamanho do *buffer* de objetos é durante o carregamento de modelo, apesar da gramática gerar procedimentalmente a cena, cuja característica pode gerar uma quantidade de objetos indefinida, os modelos CAD que usamos nesse trabalho são primordialmente estáticos. A derivação da gramática do modelo pode gerar uma variada quantidade de objetos diferentes conforme a visibilidade. Porém, o máximo de objetos gerados na cena é a quantidade de objetos que o modelo contém, que pode ser obtida no carregamento de modelo. A quantidade de objetos do modelo e a memória de vídeo disponível pela placa gráfica podem ser usados para ponderar o tamanho do *buffer* de objetos, que pode ser igual ao total de objetos do modelo, se couber na memória de vídeo disponível ou baseado em alguma heurística.

Para renderizar modelos CAD massivos com FPS interativo, é necessário descartar o máximo de objetos que não vão contribuir com a cena, o que reforça a ideia de que o *buffer* de objetos não conterá todos os objetos da cena ou grande parte da mesma. Se todos os objetos estão sempre visíveis, não faz sentido processar a gramática em CPU e transferir para GPU o tempo todo, dado que é mais eficiente fazer chamada de *draw* com os atributos já armazenadas na memória de vídeo. Logo, o mínimo de objetos será importante também para a etapa de *upload* de dados à GPU, que também deve ser o menor volume de dados possível. Assumindo que as malhas de polígonos estão em menor quantidade no modelo[5] e que cabem na placa de vídeo, ficando fixas no carregamento de modelo, a troca de dados entre CPU e GPU se resume

[5]As malhas podem estar em menor quantidade quando vista como objetos individuais, porém o volume de dados de triângulos pode ser bastante significativo em um modelo.

em enviar os escopos, poucos parâmetros extras e cores para qualquer tipo de objeto a cada quadro.

4.8.4 Balanceamento de responsabilidades entre CPU e GPU

Dentre as motivações para implementar a derivação e interpretação da gramática totalmente em CPU, temos a liberação da *thread renderer* para desenhar a cena o mais rápido possível. A *renderer* não realiza outros processamentos a não ser fazer chamadas à placa gráfica. A *thread updater* tem a responsabilidade de enviar os dados com o mínimo de volume e organizados no intuito de reduzir o *overhead* tanto do *upload* quanto dos *draw calls*. Dado que a GPU está mais livre de processamento de dados da cena, é possível incrementar o pipeline de renderização. Embora usualmente modelos CAD industriais sejam utilizados com somente cor sólida e iluminação de *phong* na renderização da cena, em certas circunstâncias pode ser desejado utilizar técnicas mais sofisticadas como renderização com textura, *Screen-Space Ambient Occlusion* (SSAO) [48], *Shadow mapping* [49], *ray-tracing*[50][6], dentre outras. Essas técnicas por sua vez vão impactar na performance da GPU.

Quanto ao *frustum culling*, existem dois momentos em que pode ser feito descarte de objetos, tanto em CPU durante a derivação e interpretação quanto em GPU no momento da renderização das superfícies. Embora durante o processamento do modelo seja oportuno descartar instâncias de objetos que estão fora do *frustum*, delegamos o descarte individual para a GPU, deixando que a CPU descarte grupo de objetos maiores agrupados nas regras de produção. Há dois motivos para essa decisão: i) Para um grupo limitado de objetos, a GPU consegue realizar o *frustum culling* de forma mais eficiente, utilizando os *tessellation shaders* como descrito na Seção 4.4, que por sua vez é realizado em paralelo; ii) Na renderização assíncrona, dado que a GPU pode está desenhando *batches* de objetos desatualizados, é interessante que estes contenham objetos próximos para diminuir o erro na renderização dos quadros, dessa forma os objetos que não foram descartados no processamento da gramática podem ser desenhados por coerência espacial.

[6]Pode ser necessário alguns ajustes na otimização da gramática do modelo, como permitir mais objetos para serem renderizados na cena. Por exemplo: o *Shadow mapping* deve considerar os objetos que estão obstruindo a luz, no caso do ray-*tracing* deve-se considerar objetos para reflexão e refração.

5
Conversão de modelos

Neste capítulo é descrito o processo de conversão de modelos CAD em MCAD *Shape grammar*. Além da conversão direta de um formato CAD para gramática, o foco principal desse capítulo é apresentar estratégias de como explorar modelagem procedimental em modelos CAD massivos baseado nos modelos reais que analisamos e as características da gramática introduzidas no Capítulo 3.

5.1
Base PDMS

Este trabalho usou modelos de uma base *PDMS* (*Plant Design Management System*), que é um sistema CAD que foi desenvolvido pela *AVEVA* [51] na década de 70. Dessa base é possível exportar arquivos no formato RVM [52] que contém geometrias do modelo, hierarquia e dados de engenharia. A base do sistema contém todas as informações integradas de um projeto, enquanto os arquivos exportados são fragmentos específicos do modelo, que o usuário pode usá-los para visualizar ou manipular em um outro software.

Nossa base de teste é composta por vários arquivos RVM's de um projeto de planta industrial que são divididos por *unidades* e tipos de *sistema*. Nesse contexto o tipo de sistema classifica sua função no processo da planta industrial, e uma unidade é composta por um conjunto desses sistemas. Criamos modelos de testes baseados nessa organização, há gramáticas de unidades individuais e da combinação de múltiplas unidades que permitem navegar na visão geral da planta industrial com todas as partes da planta integradas de uma só vez.

Uma hierarquia hipotética pode ser vista simplificadamente a seguir, onde a indentação do texto corresponde ao aninhamento dos elementos:

```
Unidade-01
  Unidade-01-CIVIL
    Detalhe-Civil
      objeto-1
      objeto-2
       ...
```

```
      ...
  Unidade-01-EQUIPAMENTOS
    ...
  Unidade-01-INSTALAÇÕES-ELÉTRICA
    ...
  Unidade-01-TUBULAÇÕES
  ...
Unidade-02
  Unidade-02-CIVIL
    ...
  Unidade-02-EQUIPAMENTOS
    ...
  ...
```

5.2 Pipeline de conversão

A Figura 5.1 mostra o pipeline de conversão que usamos para converter modelos CAD em nossa gramática. A linha superior da esquerda para direita mostra as etapas macro, enquanto a coluna de cima para baixo mostra as tarefas micro que devem ser realizadas em sua respectiva etapa.

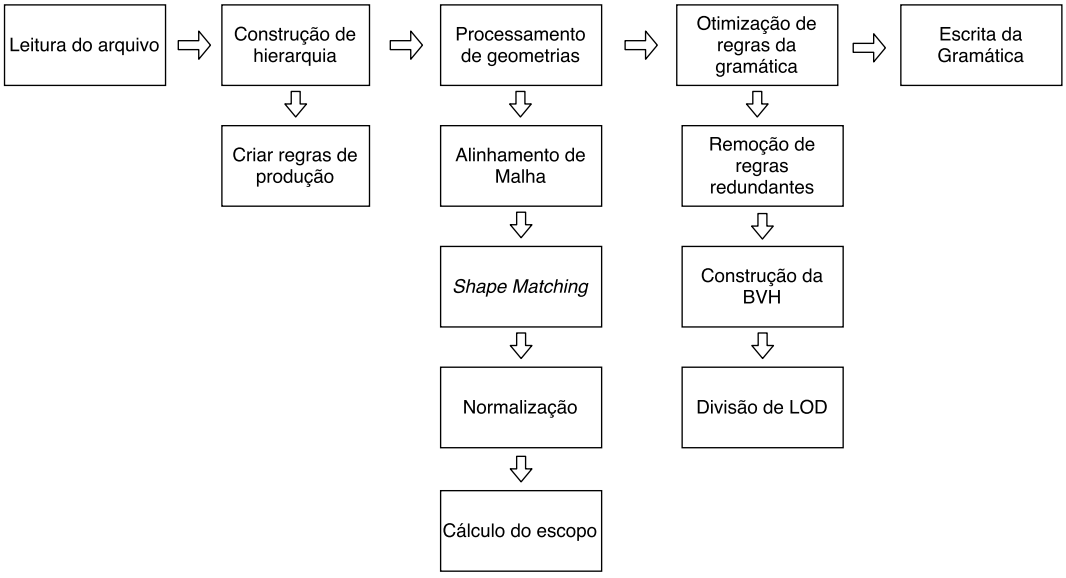

Figura 5.1: Visão geral do pipeline de conversão de modelos CAD em MCAD *Shape grammar*.

O processamento do arquivo se inicia pela leitura do arquivo e transformação da hierarquia do modelo em regras de produção. Normalmente, as geometrias dos objetos estão presentes nas folhas, que por sua vez vão ser os símbolos terminais da gramática. O formato RVM possui primitivas de alto

nível correspondentes às primitivas especializadas apresentadas na Seção 3, porém devem ser processadas para que fiquem normalizadas no formato MCAD *Shape grammar*. Na etapa de processamento de geometrias, se for encontrada uma geometria representada por uma malha de polígonos, são realizados processamentos da malha antes de ser adicionada como sucessor de sua regra de produção, detalhes desse procedimento são descritos na Seção 5.4. As malhas únicas são escritas em arquivos individuais para serem carregados durante a leitura da gramática no sistema de visualização MCAD *Shape grammar*. Após a construção da gramática, são realizados processamentos que vão aplicar otimizações das regras de produção no intuito de compactá-las e criar estruturas de aceleração para visualização do modelo. Por fim, a gramática é escrita para o sistema de arquivos junto com seus arquivos de malhas de polígonos.

5.3 Hierarquia

Durante a leitura do arquivo, o processamento dos dados é feito na forma de travessia da hierarquia em largura. A abordagem direta é transformar cada nó em uma regra de produção e colocar os nós filhos em seus sucessores. Entretanto, definimos como parâmetro da conversão uma altura máxima para aninhar a hierarquia, discutido na Seção 5.6. Em uma dada altura para um nó, todos são convertidos em sucessores do predecessor do nó de maior altura.

Seguindo a padronização de hierarquia descrita na Seção 5.1 criamos uma regra de produção que corresponde ao nó raiz de cada unidade, que por sua vez vai ter como sucessores os nós filhos imediatos que correspondem aos sistemas dessa unidade. As regras de produção de cada sistema são envolvidas com [e]. Conforme descrito na Seção 4.3, os operadores de *push* e *pop* permitem paralelização do processamento em *multithread*. A gramática a seguir mostra um exemplo de como uma gramática pode ter regras de produção processadas em paralelo:

```
Unit01 -> [ E(240.13,130.53,47.57) M(5049,2963,19) Unit01_civil ]
          [ E(199.43,99.68, 41.15) M(5031,2960,37) Unit01_pipping ]
          [ E(235.92,113.39,37.94) M(5049,2957,38) Unit01_equip ]
          [ E(231.97,113.47,24.27) M(5048,2958,31) Unit01_eletric ]
...
```

5.4 Processamento de malhas de polígonos

A cada ocorrência de geometria representada por malha de polígonos no arquivo do modelo, é realizada uma série de processamentos para otimizar a utilização dessas geometrias durante a visualização. Dentre elas, o cálculo da OBB que alinha a malha e após aplicado a normalização vai ser utilizada para criar o escopo da geometria, e o *Shape matching* que vai remover malhas redundantes que se diferem apenas pela transformação.

5.4.1 Alinhamento

Para alinhamento da malha utilizamos o PCA [53], que em processamento de geometria é uma forma simples e clássica de calcular a OBB. Primeiro, transformamos os vértices da malha de polígono no espaço do mundo, calculamos o centroide do objeto, baseado neste transladamos os vértices para o centro local do objeto. Em seguida, iniciamos o processamento do PCA utilizando como entrada os vértices da geometria. Dado V como conjunto de n vértices, a matriz de covariância C é dada por:

$$C = \frac{1}{n}(VV^T) \tag{5-1}$$

Utilizando a equação característica podemos encontrar os autovalores da matriz de covariância:

$$|C - \lambda I| = 0 \tag{5-2}$$

Se λ é um autovalor de C, I matriz identidade e temos v que como autovetor de C tal que:

$$Cv = \lambda v \tag{5-3}$$

Os autovetores de C como colunas podem construir uma matriz de rotação M que alinha os eixos dos vértices de V no espaço do mundo. Entretanto, um efeito colateral do método é que pode conter informação de cisalhamento na matriz M. Logo, é necessário realizar uma decomposição da matriz para extrair somente a rotação [54] e conservar o cisalhamento da malha, dado que não expressamos cisalhamento no escopo durante a interpretação da gramática.

5.4.2 Shape matching

Nesse trabalho aplicamos o *Shape matching* para detectar malhas de polígonos duplicados similarmente como é feito em [31]. Utilizando esta técnica podemos identificar malhas redundantes e a transformação que leva de uma a outra. Dado os vértices de uma malha m_i, estamos interessados em determinar se m_i', os vértices de

uma outra malha candidata a ser duplicada, são aproximadamente iguais para uma matriz de transformação A:

$$m'_i \approx Am_i \tag{5-4}$$

onde i é o índice do vértice. A matriz A pode ser calculada pelas matrizes derivadas:

$$A = (\sum_i m'_i m_i^T)(\sum_i m_i m_i^T)^{-1} \tag{5-5}$$

Dado que as malhas estão centralizadas, a matriz A é uma matriz 3×3 que aplica uma transformação afim de m_i em m'_i em respeito a rotação, escala e cisalhamento. Para finalizar o casamento, duas condições devem ser satisfeitas, i) o erro máximo deve ser menor que um limiar e definido como parâmetro no processo de conversão:

$$\max_{\forall i}(\|m'_i - Am_i\|) < e \tag{5-6}$$

ii) o cisalhamento deve ser nulo. A matriz deve ser decomposta [54] de tal forma que possamos extrair a rotação e escala e criar regras que vão transformar o objeto e determinar o cisalhamento para completar o casamento entre malhas.

Uma limitação evidente dessa técnica é o fato de que só realiza casamento entre malhas com número de vértices iguais e que seguem a mesma ordem. Porém, desconsiderando essa limitação, podemos otimizar a busca armazenando num mapa de *hash* as malhas únicas utilizando como índice a quantidade de vértices. Dessa forma se reduz o espaço de busca consideravelmente, acelerando o processo de conversão do modelo.

5.5 Otimização de regras

Essa seção descreve como otimizar as regras de produção para o processamento de modelos de forma mais eficiente.

5.5.1 Remoção de regras redundantes

Considere a Figura 5.2, nela as estruturas destacadas A e B, com seus escopos desenhados, mostram que um conjunto de geometrias possui um padrão bem definido. Na estrutura A são mostradas colunas representadas por cilindros com dimensões aproximadas. Em B há um padrão de combinação de caixas com tamanhos semelhantes (note as bordas do escopo) que formam a estrutura com um conjunto pequeno de transformações diferentes. Nos dois casos, percebe-se que entre um objeto e outro as rotações e escalas são preservadas, diferenciando principalmente pelas translações. As operações de escopo como apresentadas em 3.1.1 decompõem a transformação em instruções individuais. Seguindo a interpretação procedimental, podemos desenhar uma sequência de objetos preservando parte do escopo compartilhado e alterando somente a propriedade que muda de um para outro. Com essa

estratégia podemos representar uma mesma configuração de objetos de forma mais compacta utilizando operadores de escopo que uma matriz de transformação ou com as propriedades separadas.

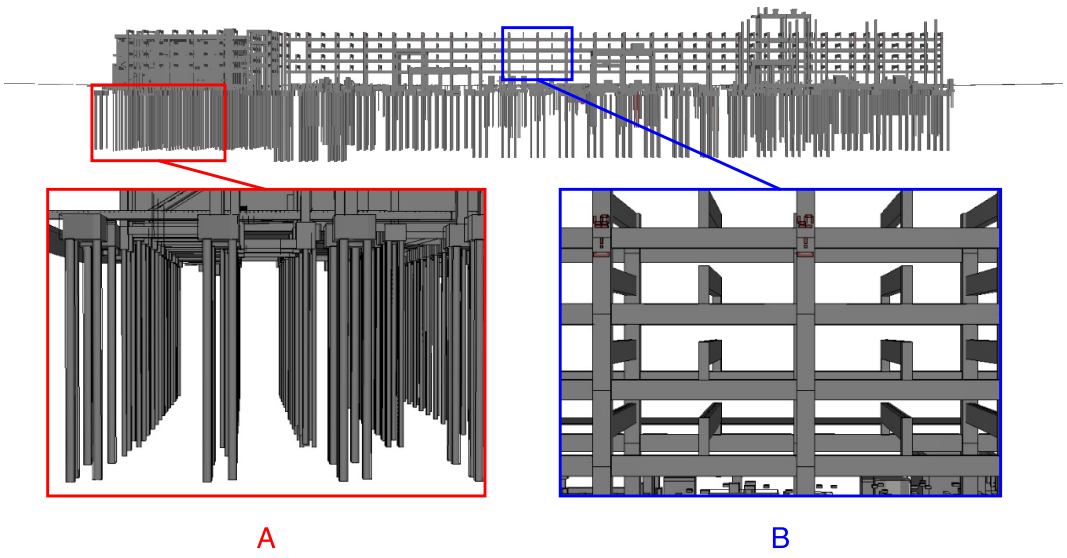

Figura 5.2: Padrões repetidos num modelo de construção civil. A. estacas da base. B. parte de um *piperack* (estrutura que suporta tubulação).

Para ilustrar uma abordagem de otimização de regras de produção, considere a gramática exemplo a seguir:

```
Base -> M(1.0, 0, 0)G(0, 0, 0)E(2, 3, 1)C(0.5, 0.5, 0.5)I("cube")
        M(1.5, 0, 0)G(0, 0, 0)E(2, 3, 1)C(0.5, 0.5, 0.5)I("cube")
        M(8.5, 0, 0)G(0, 0, 0)E(2, 3, 1)C(1.0, 1.0, 1.0)I("cube")
        M(2.5, 0, 0)G(0, 0, 0)E(2, 3, 1)C(0.5, 0.5, 0.5)I("cube")
        M(7.0, 0, 0)G(0, 0, 0)E(2, 3, 1)C(0.5, 0.5, 0.5)I("cube")
```

essa gramática foi gerada de forma ingênua gerando o escopo necessário por cada instância. Para otimizar, interpretamos essa regra durante a conversão e colocamos o resultado numa lista contendo os elementos com seu escopo. Ordena-se os elementos utilizando como teste as propriedades do escopo na seguinte ordem: rotação, escala e cor. Rotações costumam mudar pouco, escalas tendem a ser os que mais se repetem, logo essa ordenação vai servir para agrupar objetos de mesmo tamanho. Translações não são levadas em conta na ordenação, porque é parte da transformação que mais varia. A cor embora seja uma propriedade que agrupe melhor, é uma instrução de um custo menor para processar em relação as demais operações de escopo. Feita a ordenação, a nova regra de produção é dada por:

```
Base -> T(0, 0, 0)G(0, 0, 0)E(1, 3, 2)C(0.5, 0.5, 0.5)I("cube")
        T(2, 0, 0)I("cube")
```

```
T(4, 0, 0)I("cube")
T(-2, 0, 0)C(1.0, 1.0, 1.0)I("cube")
T(5, 0, 0)E(2, 3, 2)C(0.5, 0.5, 0.5)I("cube")
```

essa compressão não só diminui o consumo de memória como diminui a quantidade de regras que o interpretador deve processar durante a navegação do modelo.

5.5.2 Construção de BVH

A construção de hierarquias espaciais tem um importante papel para reduzir a complexidade do processamento do modelo. Utilizamos uma BVH para criar grupos de objetos e combinamos com a variável *scope* para descartá-los durante o processamento de uma regra de produção. A hierarquia original dos modelos já agrupa os objetos com uma moderada coerência espacial, então decidimos preservá-las. Para cada nó, calculamos sua caixa envolvente expandindo pelas caixas envolventes de seus filhos numa travessia em profundidade. Apesar da interpretação e derivação das regras de produção considerarem OBB, optamos por manter AABB na construção da BVH, dado que os modelos de nossa base foram desenhados alinhados com o eixo da origem, logo as caixas envolventes já possuem uma disposição satisfatória para realizar *frustum culling* de maneira hierárquica.

A gramática a seguir mostra como convertemos um nó interno que vai ser a caixa envolvente de outros nós.

```
grandparent(flag) -> T(10, 0, 0) E(85, 20, 30) parent(flag)
parent(flag)[flag=1] -> parent_content(1)
parent(flag)[scope>1] -> parent_content(1)
parent(flag)[scope>=0] -> parent_content(0)
parent_content(flag) -> S(0.5, 0.5, 0.5) child(flag)
                            T(0, 0 5) child(flag)
                            T(0, 0 10) child(flag)
child_1(isVisible)[flag=1] -> child_content(1)
child_1(isVisible)[scope>1] -> child_content(1)
child_1(isVisible)[scope>=0] -> child_content(0)
child_content(flag) -> ...
...
```

dado um nó *parent*, este é desdobrado em duas regras de produção: *parent* e *parent_content*, na qual a primeira tem três alternativas que vão ser avaliadas de acordo com o escopo atual e outra que gera o conteúdo do nó propriamente dito. Iniciando pela regra de produção *grandparent*, que foi derivada de seus predecessores, tem como sucessores duas operações de escopo, uma translação e uma escala que vão definir a caixa envolvente do nó, e a regra de produção *parent*. A regra *parent* vai ser avaliada recebendo como parâmetro $flag$ repassando o valor no qual

grandparent foi derivada. Nesse contexto, a variável tem como objetivo verificar se há nas derivações seguintes garantias que todos os escopos estarão contidos dentro do *frustum*, para reduzir o cálculo de visibilidade posterior na avaliação de seus sucessores. Se *grandparent* receber em *flag* o valor 1, a primeira alternativa de *parent* será selecionada que deriva *parent_content*, que por sua vez vai repassar para *child_content* o mesmo valor e assim sucessivamente. É importante escrever esse tipo de comportamento explicitamente, dado que regras de produções são genéricas e sua avaliação depende do contexto atual que pode ser dado pelo escopo ou por variáveis.

Quando $flag$ recebe outro valor, como 0 por exemplo, deve ser necessário realizar o cálculo de visibilidade que consiste principalmente em realizar o *frustum culling* no escopo atual. Se o resultado for $scope > 1$ logo *parent_content* será derivada recebendo 1 em $flag$, indicando que naquele ponto em diante todos os escopos estão contidos no *frustum*. Se $scope >= 0$, então a caixa envolvente do grupo está em intersecção, não se pode garantir a visibilidade dos objetos em seus sucessores, logo deve-se checar na hora de processá-los. Por fim quando $scope < 0$, não especificado nas regras, significa que o a caixa envolvente não está visível, como não há mais alternativas para avaliar, a derivação dessa regra de produção se encerra nesse ponto, realizando o descarte de seus possíveis sucessores.

Um cuidado que deve se tomar durante o descarte de uma derivação é em relação ao estado do escopo. Dada a natureza procedimental do processamento do modelo, o descarte de uma regra de produção cria caminhos que podem gerar escopos diferentes e influenciar o processamento das regras de produção seguintes. Nesse caso, deve-se adicionar instruções que contornem essa anomalia na derivação, como por exemplo operações de *push* e *pop* que vão preservar o escopo para a próxima regra ou usar operadores de escopo absoluto que vão garantir a configuração dos objetos independente da derivação passada.

5.5.3 Divisão de LOD

Mesmo criando BVH para realizar descartes de grupos de objetos, existem situações em que a cena inteira vai estar visível e não será possível tirar proveito da estrutura espacial por mais bem distribuída que esteja. Porém, numa visão ampliada de um modelo há alta probabilidade de muitos objetos serem descartados por *detail culling*, dada a distância que a câmera precisa manter para ter todos os objetos numa mesma visão. Exploramos essa característica para criar um esquema de LOD discreto baseado em *detail culling* utilizando a variável *lod* (Seção 4.2.2) em regras de produção. O *detail culling* reduz o volume dos *batches* de objetos que vão ser enviados a renderização, porém ainda há o custo de processar o escopo e estimar o tamanho em tela. A divisão de LOD tem o objetivo de antecipar, durante a avaliação das regras de produção, que objetos serão descartados por *detail culling* sem precisar processá-los individualmente. Considere a gramática a seguir como extensão da apresentada na Seção 5.5.2:

```
parent(flag)[flag=1] -> parent_content(1)
parent(flag)[scope>1] -> parent_content(1)
parent(flag)[scope>=0] -> parent_content(0)
parent_content(flag)[lod=0] -> parent_LOD_0(flag)
                               parent_LOD_1(flag)
                               parent_LOD_2(flag)
parent_content(flag)[lod=1] -> parent_LOD_0(flag)
                               parent_LOD_1(flag)
parent_content(flag)[lod>=2] -> parent_LOD_0(flag)
parent_LOD_0(flag) -> ...
parent_LOD_1(flag) -> ...
parent_LOD_2(flag) -> ...
```

quando a regra de produção passa dos testes de visibilidades, as próximas regras de produção vão determinar seu LOD atual conforme a visão da câmera. As regras de produção *parent_LOD_X*, onde X é o número do LOD, são subdivisões do conteúdo de *parent* que são agrupamentos de objetos baseados em tamanho. Quanto maior é o nível de detalhe do escopo (menor valor de *lod*) que precede *parent*, mais sucessores de LOD são selecionados para serem derivados.

Para construir as regras de produção que vão dividir os nós da hierarquia em LOD, elaboramos um algoritmo recursivo que faz uma travessia na hierarquia do modelo e classifica o conteúdo de cada ramificação do nó, mostrado no pseudoalgoritmo a seguir:

Algoritmo 2 Divisão de LOD

função CALCULATELOD(p, c, L)
  para todos $l \in L$ **faça**
    se $relativeSize(p, c, l) < \frac{D_c}{S_c}$ **então retorne** $\max L - l + 1$
    fim se
  fim para retorne 0
fim função

função SPLITINLOD(p, n, L)
  $l \leftarrow CalculateLOD(p, n, L)$
  se $l \neq 0$ **então**
    $splitBranch(p, n, l)$
  fim se
  $C \leftarrow children(n)$ ▷ Coleta os filhos
  para todos $c \in C$ **faça**
    $SplitInLOD(p, c, L)$ ▷ Calcula recursivamente
  fim para
fim função

A função $SplitinLOD(p, c, L)$ recebe como parâmetro um nó pai p e um de seus nós filho n em qualquer altura da árvore para um conjunto de LOD's possíveis

$L = \{0, 1, ...l_{max}\}$. Logo, o processamento é iniciado pelo nó raiz e seus filhos imediatos. A cada visita de n é classificado seu LOD em relação a p pela função $CalculateLOD(p, c, l)$, então é testado se seu LOD é o máximo, ou seja 0. Se o nó estiver fora do grupo de maior detalhe, este é remanejado para uma outra divisão da hierarquia, realizada pela função $splitBranch(p, n, l)$. A função $splitBranch(p, n, l)$ remove o nó n e cria ou atualiza uma ramificação marcada como LOD l no nó p. Essas ramificações marcadas com LOD vão ser utilizadas para criar regras de produção alternativas condicionadas pelo seu respectivo valor de LOD.

A seguir, o passo a passo de como implementar a função $relativeSize(p, c)$. Dada a Equação 2-5 que estima o tamanho da projeção da esfera conforme a distância e raio, podemos simplificar na forma a seguir:

$$s = \frac{r}{d} \tag{5-7}$$

onde s é o raio da esfera envolvente projetada na tela, r raio da esfera no espaço do mundo e d a distância da câmera para o centro do volume. Essa equação simplificada expressa proporcionalidade inversa entre distância e o tamanho da esfera em tela. Considere S_0 como raio da esfera base do LOD 0 e D_c o raio do limiar de *detail culling* que são definidos na configuração do sistema conforme a Tabela 4.2. Estamos interessados em determinar a cada LOD de nó pai, se um nó filho com tamanho s_c vai ser descartado baseado em proporção, tal que:

$$s_c < \frac{D_c}{S_0} \tag{5-8}$$

dado um nó filho com raio r_c da esfera envolvente no centro c_c e um nó pai com raio r_p e centro em c_p, a distância mínima da esfera envolvente de um nó filho para qualquer observador em função da esfera envolvendo do pai é dado por:

$$d_c = d_f - \|c_p - c_c\| + r_c \tag{5-9}$$

substituindo 5-9 em 5-7 temos:

$$s_c = \frac{r_c}{\frac{r_p}{s_p} - \|c_p - c_c\| + r_c} \tag{5-10}$$

na qual obtemos a equação que calcula a projeção da esfera de um nó filho em função das propriedades do pai. Note que $-\|c_p - c_c\| + r_c$ é um deslocamento da distância entre o nó centro do nó pai e o filho onde no pior caso a esfera envolvente do filho vai estar na borda da esfera envolvente do pai. Utilizando os valores da Tabela 4.1 podemos atribuir valores a s_p em 5-10 a cada LOD e por fim determinar se o nó filho será descartando nesse nível baseado na inequação 5-8.

Por fim, após a divisão de hierarquia de LOD, é necessário reconstruir a regra de produção considerando cada nível. Um predecessor p tem um conjunto $P = \{p_0, p_1, .., p_{l_{max}}\}$ de sucessores e um conjunto $R = \{r_0, r_1, ..., r_{lmax}\}$ de regras de produção condicional na forma $p\ [lod = i]$, onde i é o valor de LOD, logo:

$$\forall r_i, \forall p_j,\ p_j \text{ é sucessor de } r_i \ \Leftrightarrow i \geq l_{max} - j \tag{5-11}$$

A função $SplitInLOD(p, n, L)$ quando aplicada no nó da raiz recebendo seus filhos imediatos somente divide o LOD para a raiz do modelo. O processo pode ser repetido para os próximos níveis da hierarquia, inclusive nos ramos que foram criados na divisão de LOD anterior, realizando então LOD de forma hierárquica durante o processo de derivação e interpretação.

5.6 Discussão

Idealmente, para explorar o máximo da geração procedimental, os modelos deveriam ter sido desenhados utilizando regras semelhante ao que fizemos na Seção 3.2. Entretanto, utilizamos modelos pré-existentes e, como o próprio domínio sugere, que são massivos, nesse trabalho nos reservamos em elaborar regras simples, e criar um processo automatizado para gerar gramáticas e avaliar a navegação nos modelos da base que trabalhamos. Porém, as regras de conversão automáticas não são exclusivas, elas podem ser combinadas com regras manuais e vice-versa, afinal a estruturação do modelo é inteiramente feito pela gramática, que até certos níveis de abstração é legível a um humano. Uma abordagem mais sofisticada de conversão seria realizar uma modelagem procedimental inversa, semelhante ao que foi feito em [55]. Identificando regras reutilizáveis, é possível reduzir mais ainda o consumo de memória, ou usá-las para explorar os padrões do modelo de outras formas como fazer cópias de estruturas baseado em *templates* identificados.

Embora possamos expressar a hierarquia inteira do modelo em gramática, definimos uma altura máxima para conversão. A altura da árvore fixada se adapta melhor às otimizações da remoção de regras redundantes (Seção 5.5) e divisão de LOD (Seção 5.5.3). Na remoção de regras redundantes, é mais conveniente ter os objetos semelhantes no mesmo nível da árvore, o que vai depender da estruturação do modelo, a redução da altura aumenta a probabilidade dessa configuração acontecer. Outro problema na divisão de LOD como apresentamos, é a pulverização da hierarquia, dado que pode ser feito recursivo entre todos os nós da árvore, o que pode gerar um consumo de memória proibitivo.

Na Inequação 5-6, existe o erro e que determina se duas malhas de polígonos casam para serem compartilhadas como uma mesma geometria. Para definir esse parâmetro, o erro tolerável vai depender do projeto da base e sua respectiva escala do modelo. Em nossa instância, 1 unidade do mundo 3D é equivalente a 1 metro, atribuímos esse erro $1e-3$, que vai definir a precisão de 1 milímetro a cada vértice das malhas de polígonos, o que acreditamos ser razoável para esses modelos.

Na construção do LOD de uma regra de produção utilizamos os parâmetros D_c e S_0 vindo da Tabela 4.2 para criar o conjunto de sucessores que vão ser descartados conforme o LOD se altera durante a navegação. Assumindo que a imagem gerada após o detail *culling* de uma cena é aceitável, nossa estratégia de LOD é conservadora. Dadas as esferas envolventes entre o filho e pai, quando a esfera do filho está na borda do pai, podemos ter o pior caso e melhor caso em relação ao

observador. No pior caso, o filho está na menor distância entre a câmera e o centro do pai, no melhor caso o filho está no lado oposto, o que já permitiria descartá-lo muito previamente. Se os requisitos permitirem uma abordagem mais branda, o limiar utilizado na Equação 5-8 poderia ser maior, pois em determinadas visões esses objetos vão ter alta probabilidade de serem descartados ou de estarem obstruídos, uma vez que são objetos proporcionalmente bem menores no conjunto em que fazem parte.

Uma clara vantagem de utilizar a abordagem procedimental para realizar LOD, é o fato de podermos reutilizar regras de produção. A abordagem de LOD discreto clássica produziria um consumo de memória extra não viável para modelos massivos. Usando a divisão de LOD como fizemos, só precisamos ligar e desligar um conjunto de objetos conforme a visão da câmara, desenhar o LOD máximo por exemplo só consiste em enviar todos os níveis de detalhes, e conforme for diminuindo o tamanho do conjunto do todo, os demais detalhes vão sendo colapsados gradativamente. Os níveis de detalhes podem ser aninhados e combinados com os nós da BVH, o que contribui para dividir complexidade do modelo significativamente.

6
Avaliação

Esse capítulo descreve a avaliação do MCAD *Shape grammar* aplicado no domínio de modelos CAD massivos sob os aspectos principais que trabalhamos durante a pesquisa, como consumo de memória e renderização em tempo real.

6.1
Tecnologias e ambiente de *benchmark*

Todo o código foi desenvolvido utilizando a linguagem C++17 [56]. Dividimos a implementação em dois projetos: *engine* e *conversor*. A *engine*, pelos requisitos que precisa cumprir para visualizar modelos massivos, possui uma implementação mais sofisticada, seguindo boas práticas de programação para manter alta performance e gerenciamento de memória. A implementação do conversor, por ser off-line, é mais despretensiosa, pois o objetivo é apenas que cumpra seu papel no tempo que for necessário, e os tempos de conversão são apresentados apenas para referência. Para cálculos matemáticos em CPU utilizamos a biblioteca GLM [57], que possui um papel importante para boa performance da *engine* dado que esta implementa funções de álgebra básica utilizando SIMD (*Single Instruction Multiple Data*) quando disponível no *hardware*. A utilização de SIMD no processamento acelera o cálculo de álgebra comum como operações de matrizes e vetores, cuja a aplicação é bastante utilizada na implementação das técnicas descritas no Capítulo 2. Como API de acesso a placa gráfica utilizamos OpenGL 4.1 e GLSL 4.1 [58].

Nas avaliações do desempenho da *engine* MCAD *Shape grammar*, utilizamos uma máquina com as seguintes configurações:

- **Processador**: Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
- **Memória principal**: 6GB RAM
- **Placa de vídeo**: Geforce GTX 760 2GB VRAM @ 980Mhz
- **Sistema Operacional**: Linux Ubuntu 16.04 64bits

6.2
Modelos

Criamos uma base de testes formada por 4 modelos, cada um com nível de complexidade diferente para avaliarmos nossa proposta em cenários variados. Rotulamos os modelos como: *M1*, *M2*, *M3* e *M4*. Cada modelo contém no mínimo

Tipos de objetos	M1	M2	M3	M4
Caixas	112.279	248.276	808.279	2.974.386
Cilindros	160.185	389.589	1.244.624	3.833.138
Cones	16.122	36.987	130.512	366.781
Toros	16.622	53.931	156.555	501.913
Semiesferas	2.510	6.858	29.898	97.665
Esferas	284	1353	2.280	8.926
Malhas de polígonos	168.106	192.753	859.831	3.472.876
Objetos paramétricos	308.002	736.994	2.372.148	7.782.809
Total de objetos	**476.108**	**929.747**	**3.231.979**	**11.255.685**

Tabela 6.1: Estatística dos tipos de objetos dos modelos.

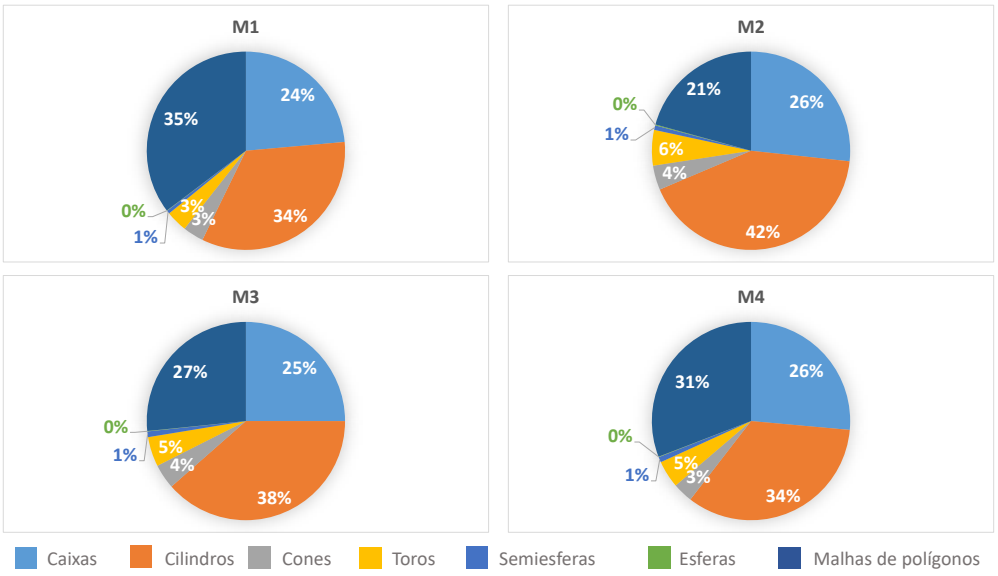

Figura 6.1: Percentual de tipos de objetos por modelo.

uma unidade e todos os sistemas que a compõem. Uma descrição breve dos modelos a seguir:

- **M1**, formado por uma unidade média.
- **M2**, formado por uma unidade grande, uma das maiores do projeto.
- **M3**, formado por 5 unidades, contém M1 e M2 e mais 3 que são as maiores unidades do projeto.
- **M4**, formado por 116 unidades, contém todos os modelos da base de um projeto de uma planta industrial.

A Tabela 6.1 mostra a quantidade de objetos por tipo dos modelos.

A Figura 6.1 ilustra a distribuição em percentagem dos tipos de objetos. O M4 contém todos os objetos da base, logo podemos ver que 69% dos objetos são paramétricos e 31% são malhas de polígonos. A maioria dos objetos são do tipo cilindro, e as semiesferas e esferas são os mais raros.

	M1	**M2**	**M3**	**M4**
Somente Malhas	4.72M	11.3M	33.7M	164M
Resolução 16×16	23.1M	59.4M	184M	638M
Resolução 32×32	49.6M	135M	415M	**1.37B**

Tabela 6.2: Sumarização de triângulos das malhas de polígonos em diferentes resoluções das superfícies paramétricas.

Como vemos, dado que a maioria dos objetos são paramétricos, a quantidade total de triângulos vai depender da resolução das superfícies paramétricas. A Tabela 6.2 mostra a quantidade de triângulos dos modelos dependendo da resolução da tecelagem aplicada. A resolução da tecelagem é definida por $R_x \times R_y$, onde R_x e R_y são as quantidade de divisões no eixo x e y respectivamente de uma malha a ser "dobrada" utilizando equações paramétricas para gerar a superfície paramétrica. Podemos ver que o modelo M4 pode chegar a mais de um bilhão de triângulos.

Por fim, a Figura 6.2 mostra a visão geral de todos os modelos citados anteriormente. Nessa imagem enfatizamos a comparação de escala e complexidade entre cada modelo.

6.3 Dados de Conversão

Como descrito no Capítulo 5 trabalhamos na conversão de uma base PDMS exportada como arquivos RVM's. Um dos parâmetros que precisamos definir na conversão dos modelos em gramática é a altura da árvore dos modelos. Definimos esse parâmetro experimentalmente como 4, onde o primeiro nível é a unidade que agrupa objetos com alta coerência espacial pelo relacionamento semântico que tem entre si, o segundo são seus sistemas e adicionalmente mais dois níveis de agrupamentos de objetos. O limiar de *detail culling* foi de 10 $pixels^2$ para divisão de LOD, parâmetro que será discutido com mais detalhe na Seção 6.4.1. As gramáticas geradas pela conversão foram armazenadas em formato binário.

A Tabela 6.3 mostra dados gerais da conversão para fins de referência, onde cada arquivo contém os dados de um sistema específico de uma unidade. Não usamos o tamanho dos arquivos originais para comparações pois contém dados de hierarquia e metadados que não convertemos em gramática. Logo, fizemos os cálculos do consumo de memória mais adequado baseado em outras representações genéricas que utiliza os dados convertidos, sendo até mais compacto que o original apresentado na tabela. Destacamos o tempo de conversão de M4 porque durante a conversão detectamos que o programa consumiu mais memória que o disponível na máquina, forçando a paginação do sistema operacional e consequentemente aumentando o tempo de processamento do modelo.

A Tabela 6.3 mostra os resultados da conversão dos modelos no formato da gramática. A parte superior da tabela mostra as quantidades de regras de produção e símbolos gerados para os modelos. As quantidades de símbolos englobam

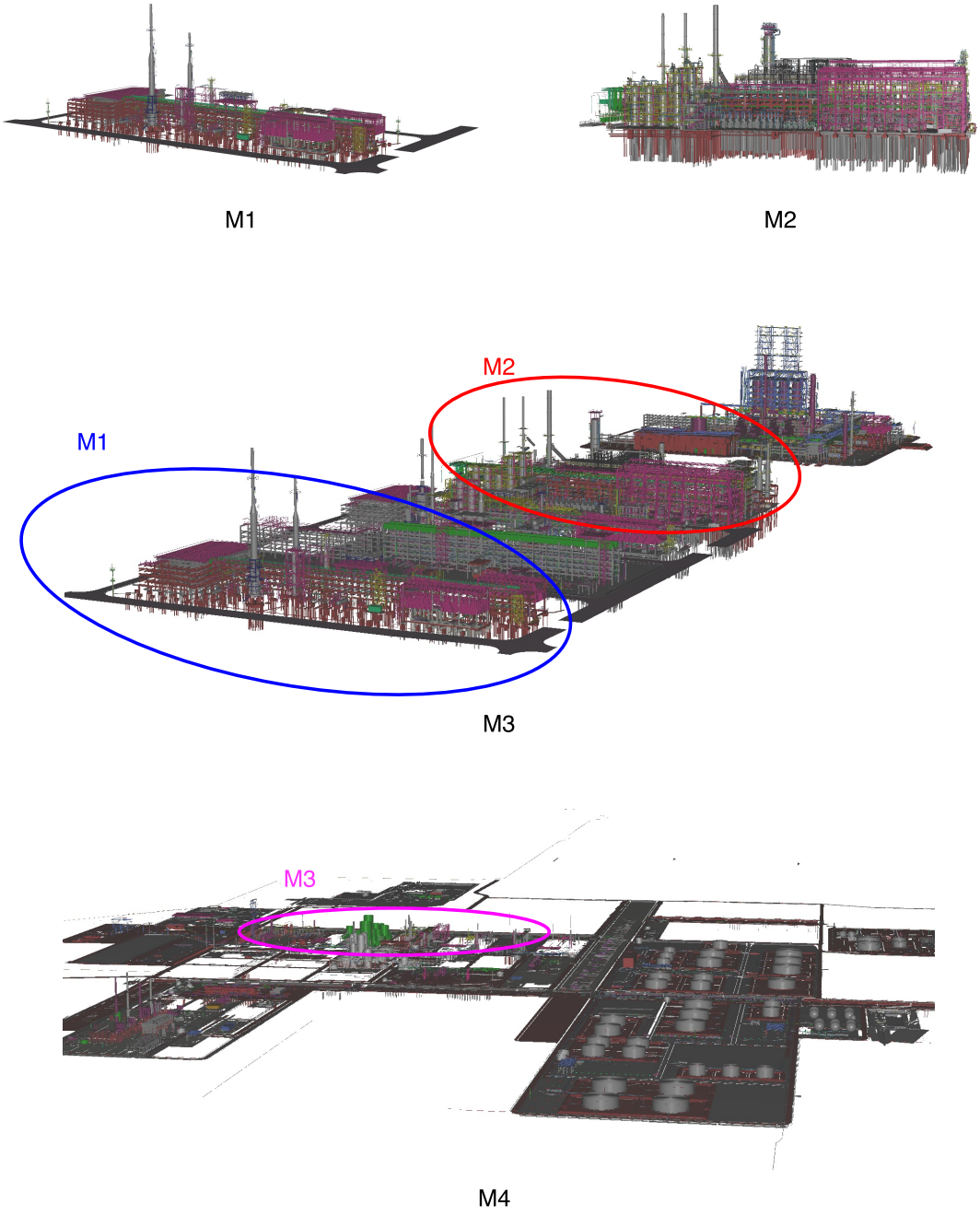

Figura 6.2: Todos os modelos da base de dados, em colorido o posicionamento de cada modelo quando um contém o outro.

	M1	**M2**	**M3**	**M4**
Arquivos convertidos	9	8	41	982
Tamanho original dos arquivos	308MB	680MB	2,2GB	11,2GB
Tempo de conversão	42s	2,85min	23min	**143min**

símbolos terminais e não terminais, que considera símbolos replicados na contagem. Na definição da gramática esses símbolos são apresentados em um conjunto, ou seja, sem ordem ou replicação, entretanto estamos interessados em mostrar na prática o volume de dados que um modelo se transforma quando representado como gramática. A divisão seguinte da tabela mostra o percentual de escalas, rotações e cores redundantes que puderam ser suprimidas pelo aproveitamento do escopo entre a geração de uma instância para outra. A parte mais inferior da tabela mostra os resultados referentes ao processamento de malhas utilizando *Shape matching* que detecta malhas duplicadas e permite que uma geometria seja compartilhada por diferentes operadores de instância. Por fim, a última linha mostra o total de memória necessária para armazenar cada modelo inteiro em MCAD *Shape grammar*.

	M1	**M2**	**M3**	**M4**
Símbolos	1,33M	2,84M	9,55M	30,1M
Regras de produção	10,7K	26,6K	80K	216K
Escalas redundantes (%)	54%	39,6%	44%	56,3%
Rotações redundantes (%)	75,8%	67,9%	71,6%	77,2%
Cores redundantes (%)	98%	97,3%	97,6%	98%
Consumo das regras de produção	13,2MB	28,4MB	94,4MB	298,6MB
Malhas únicas	3,48K	10,5K	38,2K	101K
Malhas redundantes (%)	99,2%	95.4%	97,7%	98,6%
Consumo total das malhas	238MB	562MB	1,42GB	8,17GB
Consumo malhas únicas	11,2MB	95,4MB	198,8MB	1,02GB
Consumo total da gramática (regras de produção + geometrias)	**24,4MB**	**123,8MB**	**293,2MB**	**1,31GB**

Tabela 6.3: Resultado da conversão dos modelos testes em MCAD *Shape grammar*.

Para fins de comparações, calculamos a quantidade *bytes* necessários para armazenar as transformações dos objetos usando duas abordagens: uma matriz 4×4 (16×4 *bytes* por *float*) e atributos decompostos em rotação em Euler, translação e escala (3×4 *bytes* por *float*), além desses atributos consideramos que no mínimo cada objeto tem um identificador (inteiro de 4 *bytes*) e uma cor (inteiro de 4 *bytes*). A Figura 6.3 mostra um gráfico comparando o consumo de memória dessas descrições mínimas de cena com a de MCAD *Shape grammar* contendo todas as regras de otimizações (BVH e LOD).

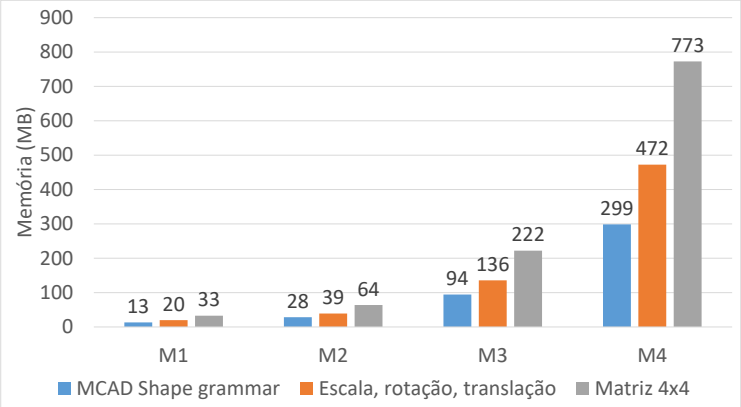

Figura 6.3: Comparação entre diferentes abordagens de armazenamento de transformações de objetos.

6.4 Performance da Engine

Esta seção avalia uma *engine* que implementa a engine MCAD *Shape grammar* proposto no Capítulo 4 com diferentes parâmetros de configuração e em diferentes aspectos. Na Seção 6.4.2 avaliamos a renderização das superfícies paramétricas isolada do processamento da gramática em CPU. Na Seção 6.4.3 testamos a engine completa (CPU e GPU) utilizando os modelos de testes de nossa base.

6.4.1 Configuração do sistema

A Tabela 4.2 apresenta os parâmetros que precisam ser definidos para configurar a *engine*. Em nossos testes combinamos alguns desses parâmetros que estão sumarizados na Tabela 6.4. O tamanho base do LOD 0 definimos experimentalmente, de tal forma que o LOD contínuo das superfícies paramétricas se mantivesse suave por todo o LOD contínuo. Os limiares de *detail culling* foram definidos com base do LOD 0, calculamos o raio da esfera envolvente em função de sua área e tiramos a percentagem desse raio para definir a área de limiar do *detail culling*. As percentagens que usamos foram 1%, 2% e 3% cujas as áreas estão na Tabela 6.4, e vamos rotulá-las dessa forma quando formos referenciar o valor desse parâmetro. Definimos o tamanho do *buffer* de *batches* como 50MB que é suficiente para mais de um milhão de objetos, que foi um limite superior que identificamos testando a navegação no modelo M4 para resolução *FULL HD* com limiar de 1%.

A Figura 6.4 mostra o impacto da configuração de *detail culling* em um cenário médio. Há valores em que o impacto é quase imperceptível e outros casos mais severos que vão ser discutidos mais à frente. Uma vez que usamos a área em pixels para descartar objetos, o tamanho da tela vai influenciar diretamente na performance da *engine*, além do fato de que uma câmera com maior abertura vai aumentar o tamanho do *frustum* incluindo mais objetos para renderização. Então, diversificamos alguns

Parâmetro	Valor
Tipo de renderização	Síncrona e assíncrona
Tamanho do *buffer* de *batches*	50MB
Quantidade de *workers*	0, 2, 4 e 8
Limiar de detail *culling*	10, 41 e 93 $pixels^2$
Tamanho base do LOD 0	103K $pixels^2$

Tabela 6.4: Parâmetros da *engine* utilizados nos testes.

testes alterando a resolução da tela. Definimos como resolução padrão 960×720, no qual focamos maior parte dos testes, as demais resoluções são enumeradas a seguir:

- 640×480, aspecto padrão baixa resolução (480p);
- 960×720, aspecto padrão alta resolução (nosso padrão);
- 1280×720, aspecto *wide* HD (720p);
- 1920×1080, aspecto *wide full* HD (1080p).

6.4.2 Renderização de superfícies paramétricas

A seguir, apresentamos a avaliação da renderização das primitivas especializadas que utilizam equações paramétricas em *tessellation shaders*. Nos testes desativamos a derivação e interpretação *on-the-fly* em CPU, realizando uma única vez e enviando os dados para a GPU, coletando somente os dados da renderização em uma tela com resolução 960×720. Criamos três cenários de testes com magnitude de objetos diferentes: 62,5K (250×250), 250K (500×500) e 1M (1000×1000) de objetos. Os cenários foram criados com a gramática a seguir, que distribui objetos uniformemente em um plano nas dimensões X e Y:

```
Test62k -> S(500, 500, 2) Repeat("XY", 62500){object}
Test250K -> S(500, 500, 1) Repeat("XY", 250000){object}
Test1M -> S(500, 500, 0.5) Repeat("XY", 1000000){object}
object ->  S(0.8, 0.8, 0.8) I("<object>")
```

as regras de produção *Test62K*, *Test250K* e *Test1M* são as regras que constroem os seus respectivos cenários mencionados anteriormente. A última regra de produção *object* contém o operador instância, o qual trocamos pelo nome do tipo de objeto que avaliamos em cada teste. O teste consiste em realizar um caminho de câmera que se inicia pelo ponto A até D, no qual a câmera se move a 30 *unidades/segundos*, executando o percurso em aproximadamente 21 segundos. A Figura 6.5 mostra o desenho esquemático do cenário de teste e alguns quadros chaves do caminho da câmera para o teste com 62,5K objetos.

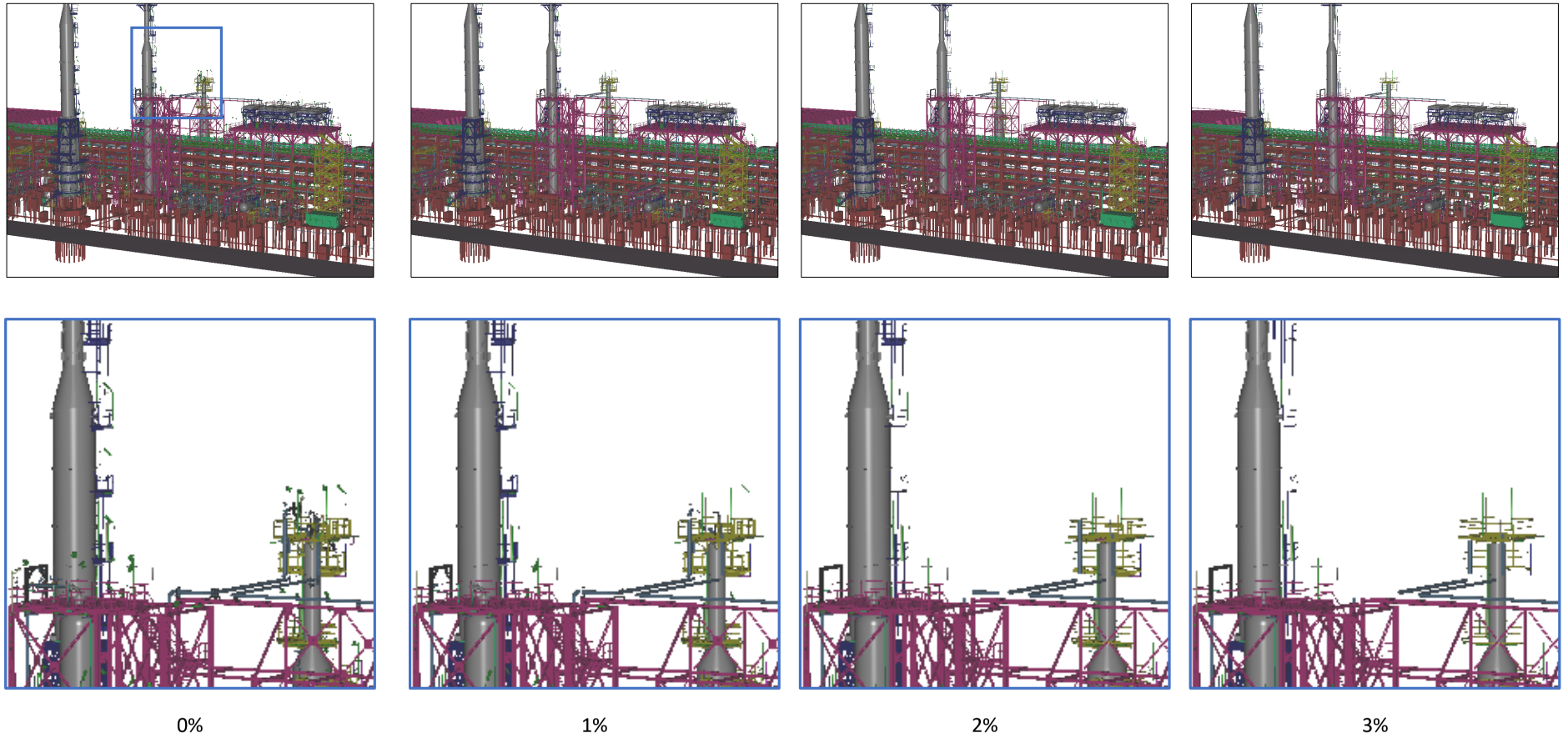

Figura 6.4: Quadros de uma cena com diferentes limiares de *detail culling*, onde 0% indica o máximo de detalhe. Acima, visão geral. Abaixo, detalhe em destaque azul ampliado. Imagens geradas na resolução 960×720.

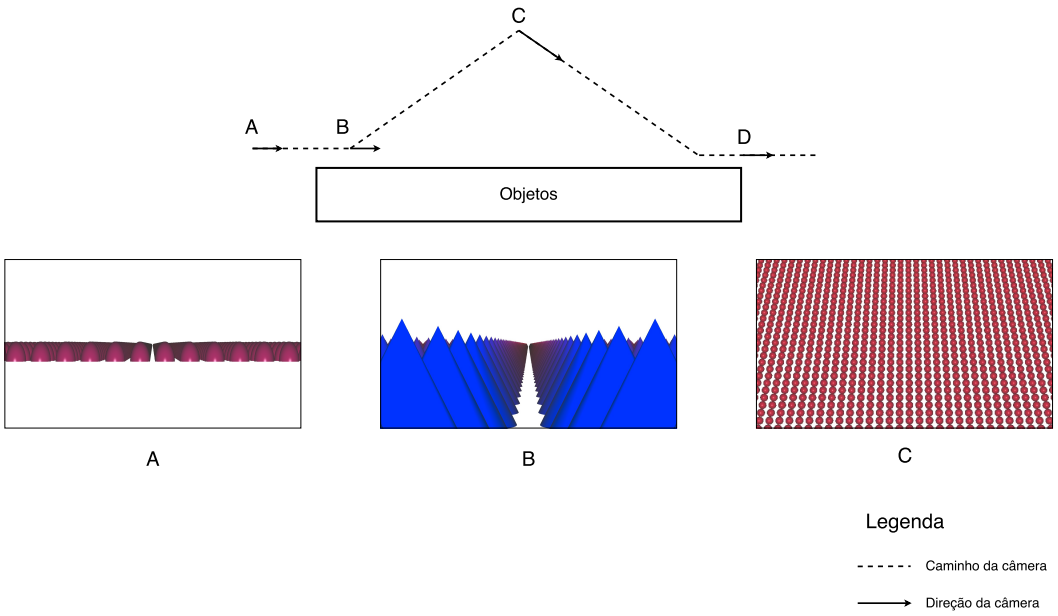

Figura 6.5: Acima, esquema do cenário para o caso de teste de renderização de superfícies paramétricas em uma visão lateral. As letras A, B, C, D correspondem a quadros chaves que podem ser vistos nas imagens abaixo para 62,5k objetos, com exceção de D que não mostra nenhum objeto na imagem. As cores indicam o LOD contínuo atual do objeto onde do azul para vermelho correspondem maior e menor nível de detalhe respectivamente.

Para cada cenário testamos com os objetos paramétricos em diferentes configurações: malhas de triângulos com tecelagem feita em CPU utilizando resolução 16×16 e 32×32 que possuem um bom balanceamento entre renderização e suavização da superfície para ser renderizados com instanciação; e tecelagem diretamente em GPU como descrito na Seção 4.4. No caso da tecelagem em GPU utilizamos a resolução 128×128 e 256×256 no LOD base.

A Figura 6.6 mostra em gráficos a comparação da média de FPS para cada tipo de objeto e suas configurações no cenário mostrado na Figura 6.5. Em geral, na abordagem de instanciação o FPS é constante mesmo no ponto D do caminho de câmera, quando não há objetos visíveis. Por outro lado, o FPS da renderização de superfícies paramétricas em GPU vai sofrer variações de acordo com a visão como pode ser visto no gráfico da Figura 6.7 que será descrito em mais detalhes mais a frente. Para o cenário de 62,5k e 250K objetos, todas as abordagens mantêm taxas interativas de visualização, entretanto no cenário de 1M de objetos somente a implementação com *tessellation shaders* manteve FPS navegável. Percebemos que a resolução das malhas impactou muito pouco na renderização das primitivas especializadas, na maioria dos cenários mantiveram o mesmo valor mesmo para 1M de objetos. Em todos os testes a renderização das superfícies paramétricas utilizando *tessellation shaders* foi mais eficiente que utilizando instanciação.

A Figura 6.7 mostra o gráfico que plota o FPS ao longo do tempo do caminho

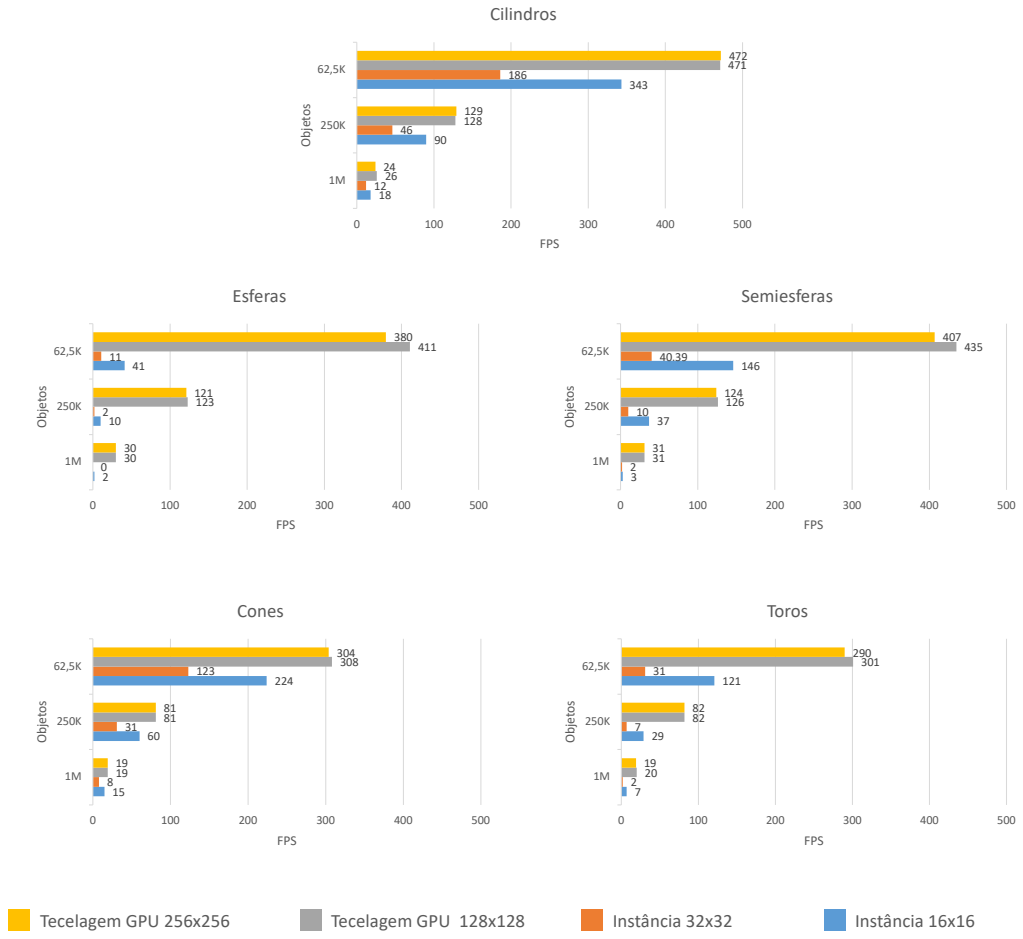

Figura 6.6: Média de FPS da renderização de objetos paramétricos em diferentes abordagens e resolução de malha de polígonos.

de câmera para o cenário de 1M de cilindros com resolução 256×256. Para essa amostragem fizemos a decomposição da renderização ativando e desativando o *frustum culling* e LOD no intuito de analisar a efetividade das duas técnicas implementadas em GPU. As duas técnicas isoladas têm FPS bem mais baixo que quando combinadas, sendo o LOD a mais baixa de todas. As letras e linhas tracejadas são aproximadamente os pontos chaves do caminho de câmera correspondente da Figura 6.5.

6.4.3 Testes com os modelos

Nesta seção avaliamos a implementação da engine completa, com interpretação e derivação *on-the-fly* em CPU combinados com renderização de superfícies paramétricas utilizando *tessellation shaders* em GPU. Para cada modelo da base criamos um percurso de câmera que simula circunstâncias diferentes que a *engine* deve lidar baseado na regras de produção da gramática que foram geradas dos modelos. Nos testes, mesmo mudando o limiar de *detail culling*, fomos conservadores e

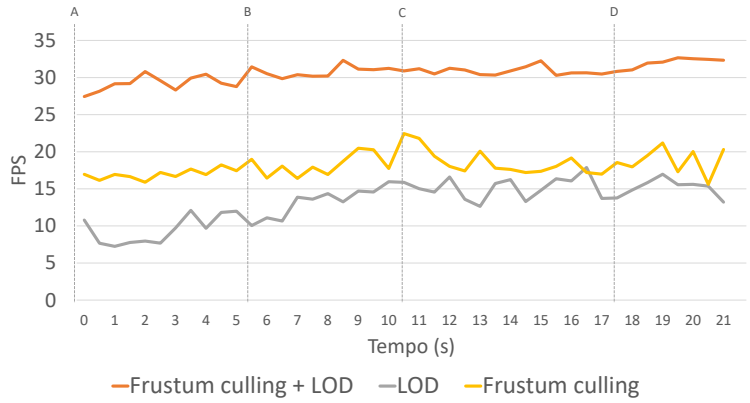

Figura 6.7: Gráfico de comparação da renderização de superfície paramétricas utilizando *tessellation shaders* com o *frustum culling* e LOD ativados separadamente. Os resultados foram obtidos no cenário com 1M de cilindros.

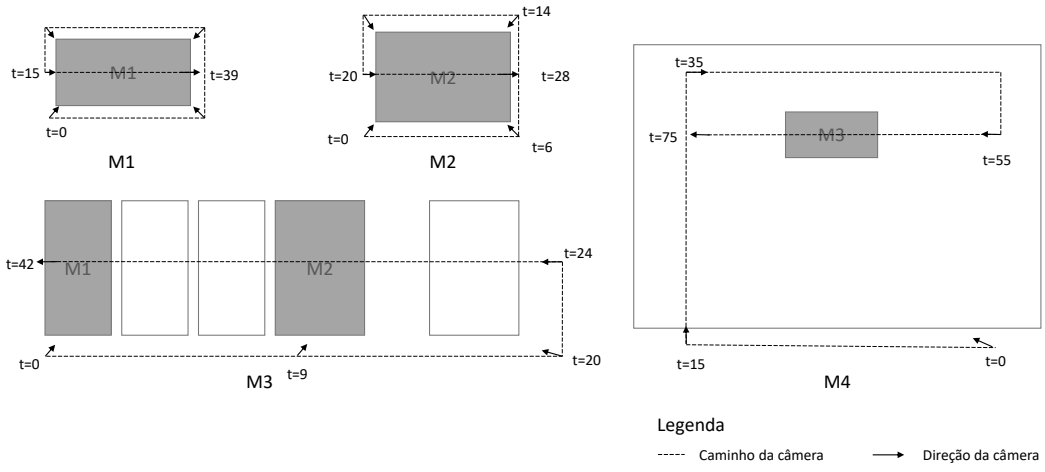

Figura 6.8: Esquemas dos caminhos da câmera sob os modelos executados durante os testes, onde t é o instante em segundos aproximado do percurso.

utilizamos uma única versão de modelo que foi convertido considerando o limiar de 1%. A Figura 6.8 mostra o percurso executado nos modelos durante os testes com alguns instantes aproximados para servir referência nos gráficos gerados.

A Figura 6.9 mostra para cada modelo dois gráficos na renderização síncrona: o FPS, que conta também com o tempo de *upload* dos dados, ao longo do tempo e o da direita a quantidade de objetos gerados por quadro. Nos gráficos há a comparação para os diferentes limiares de *detail culling* para a resolução padrão (960×720). Podemos perceber que as quantidades de objetos gerados e o FPS são inversamente proporcionais, como esperado. Entre os limiares de 1% e 2% a quantidade de objetos gerados alteram significativamente mais do que entre 2% e 3%, entretanto os FPS's desses limiares mantiveram-se proporcionais. Essa desproporção provavelmente se deve ao fato de que esses objetos da diferença entre 1% e 2% possuem um custo bem baixo de serem renderizados, principalmente por causa da tecelagem em GPU, logo

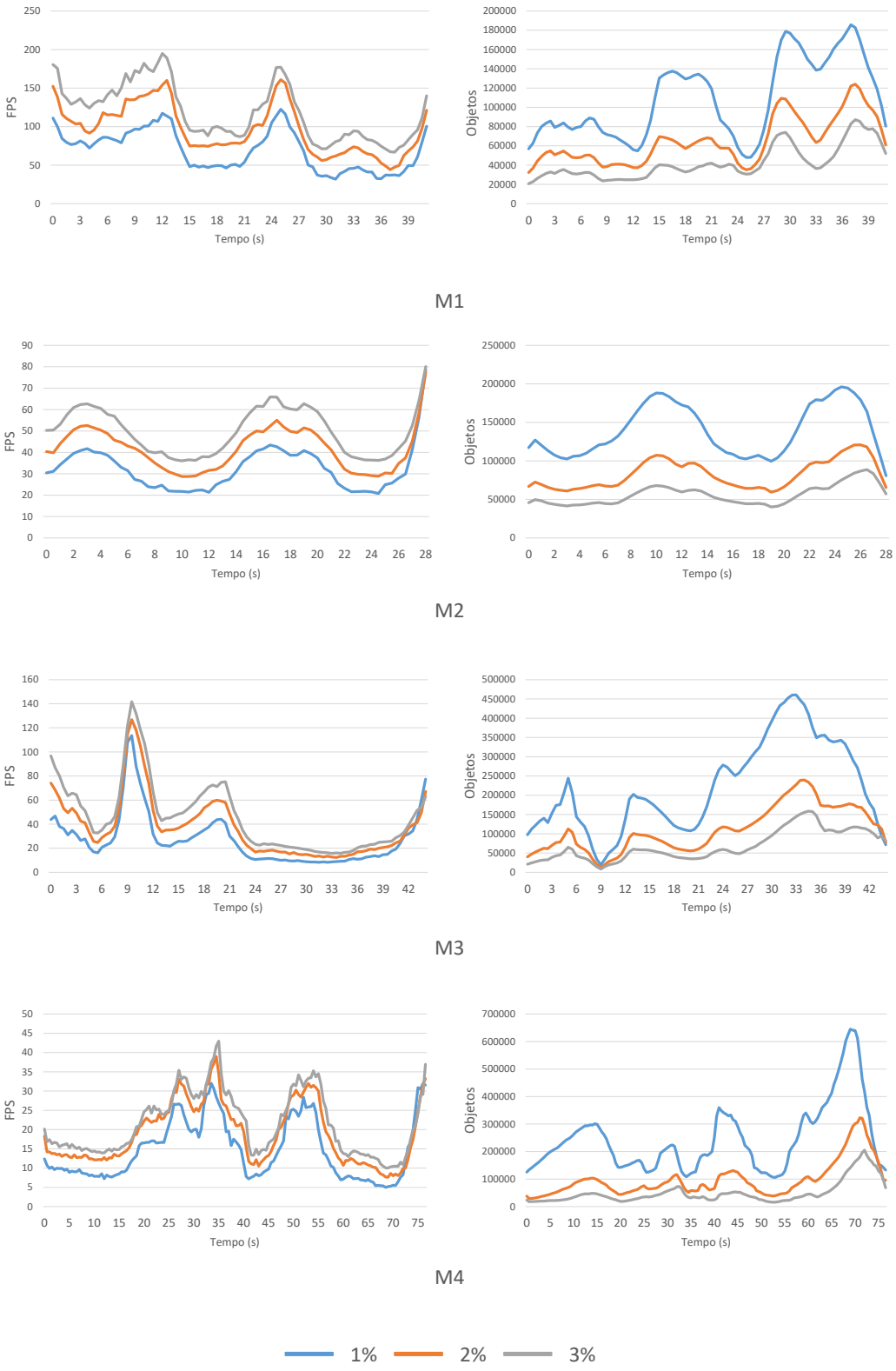

Figura 6.9: No lado esquerdo, gráficos dos FPS's ao longo do tempo para os quatro modelos na renderização síncrona. No lado direito, quantidades de objetos enviados para renderização ao longo do tempo.

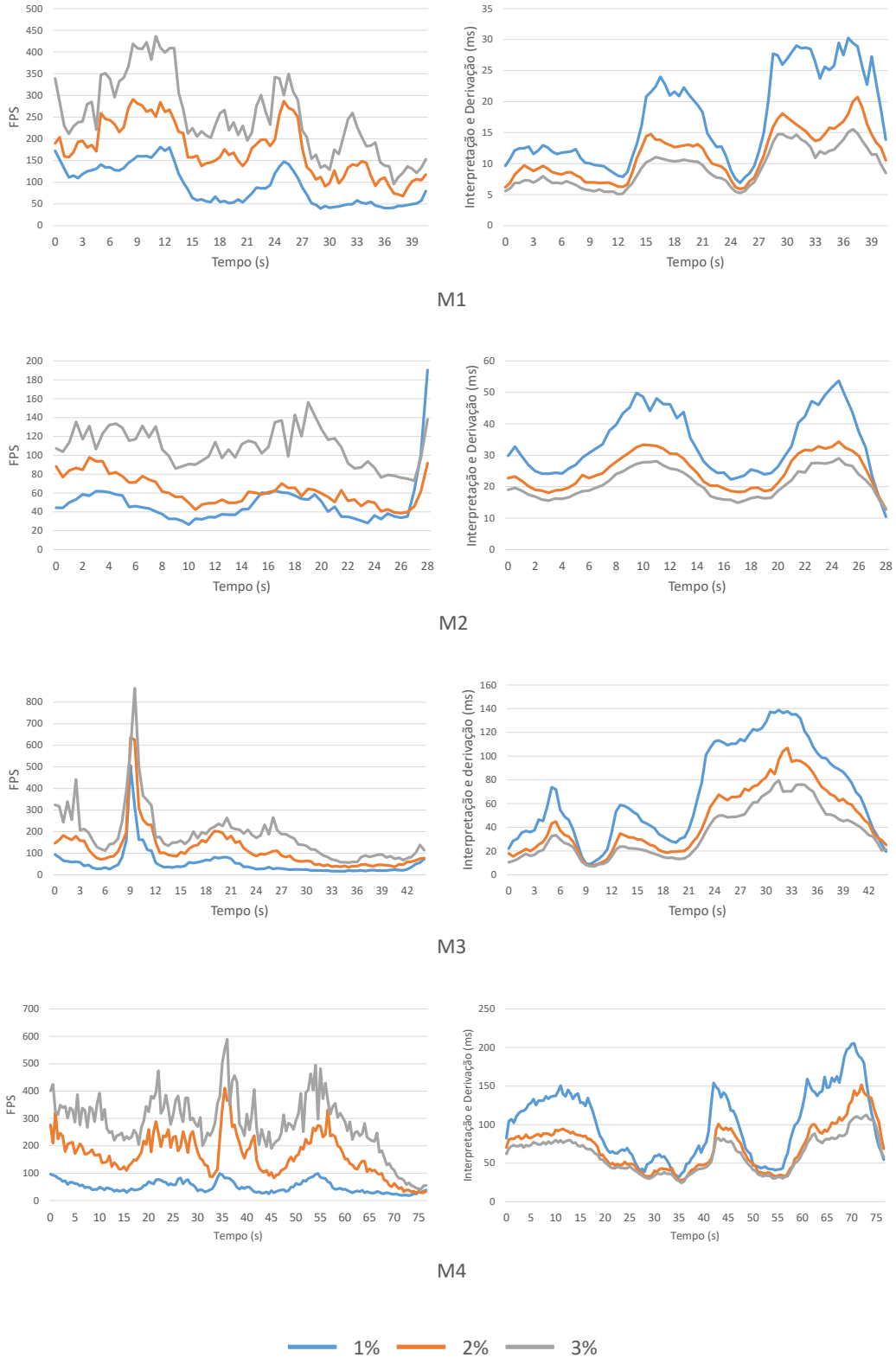

Figura 6.10: No lado esquerdo, gráficos dos FPS's ao longo do tempo para os quatro modelos na renderização assíncrona. No lado direito, tempo de interpretação e derivação em milissegundos.

a economia se deu mais pela redução de *overhead* na transferência de dados entre CPU e GPU. Para os modelos M1 e M2, os FPS's geram uma navegação bem fluida mesmo para o menor limiar, acima de 20 FPS e a partir de 2% o FPS é 30 que é a taxa padrão ideal para visualização em tempo real em cenas 3D. Para os modelos M3 e M4, na abordagem síncrona a queda de FPS é bem mais significativa, próximo de 5 FPS no instante $t = 66s$ para o modelo M4 com limiar de 1%, entretanto para os maiores limiares o mínimo foi de aproximadamente 10 FPS, que embora não seja fluido ainda é uma taxa que mantém interatividade na navegação.

A Figura 6.10 mostra os gráficos de FPS para renderização assíncrona. Na esquerda há o FPS ao longo do tempo e na direita o tempo de derivação e interpretação *on-the-fly* executado na CPU em milissegundos. No geral, o FPS foi bem mais alto em todos os cenários comparados a renderização síncrona em todos os modelos gerando sempre uma navegação fluida. O tempo máximo de quadros "errados" foi de $200ms$ no modelo M4 quando aplicado o limiar de 1% no instante $t = 70s$.

Considerando que o tempo de processamento da gramática em CPU e que a quantidade objetos gerados são equivalentes entre síncrona e assíncrona, a derivação e interpretação *on-the-fly* da gramática tem um impacto diferente no FPS da renderização assíncrona em relação a síncrona. Na abordagem síncrona, a cada quadro a *thread* de *renderer* deve aguardar *batches* novos antes de renderizar, o que não ocorre na assíncrona. Logo, quando o processamento na *thread updater* está engargalado com trechos custosos da cena, na *thread renderer* haverá ciclos que não têm *batches* novos para consumir. Durante esses pontos a renderização vai ser significativamente acelerada, pois os dados já estão na memória da placa gráfica e o ciclo só vai consistir em executar chamada de desenho à GPU. Esse efeito podemos ver nos gráficos de M1 e M2 da Figura 6.10, embora sejam modelos menores para renderizar, os FPS's não são tão superiores em relação a renderização síncrona quando comparadas ao ganho que foi nos modelos M3 e M4. Os modelos M1 e M2 têm gramáticas mais rápidas de serem processadas o que faz com que a renderização e processamento ocorram quase simultaneamente, se comportando como a renderização síncrona. Outra evidência da anomalia pode ser vista após o instante $t = 70s$ do percurso de M4, a quantidade de objetos e o tempo de CPU sofrem uma queda, na renderização síncrona o FPS aumenta significativamente, entretanto na renderização assíncrona o FPS cai um pouco e se mantém em inércia.

A Figura 6.11 mostra um gráfico que plota o tempo de processamento da gramática na CPU para o modelo M2 com limiar de detail culling de 1%. A utilização de múltiplas *threads* é bastante efetiva para acelerar a geração da cena, reduzindo o tempo por mais da metade quando comparado ao tempo de uma única *thread*. Conforme o número de *threads* aumenta, o tempo de processamento começa a convergir para um tempo só, entre 6 e 8 *threads* não houve diferenças tão significativas. Atribuímos essa limitação ao fato de que as regras de produção não são balanceadas para processamento paralelo, aliado ao escalonamento de threads

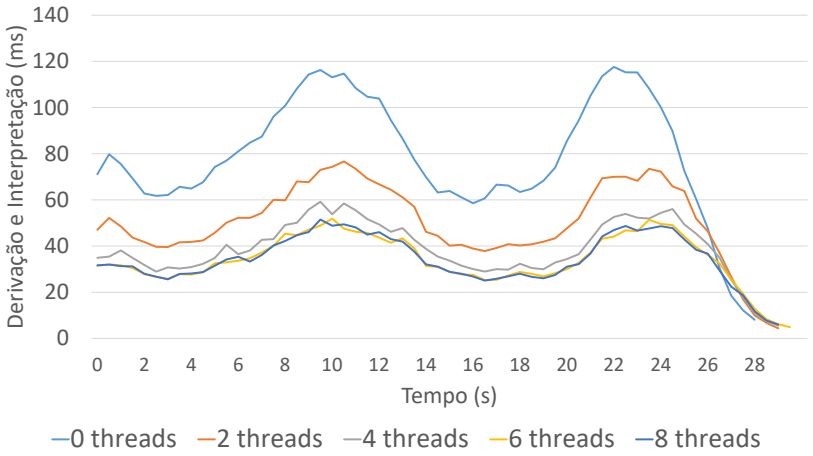

Figura 6.11: Gráfico do tempo de derivação e interpretação ao longo do tempo para o modelo M2 com diferentes números de threads para o limiar de detail culling de 1%.

	M1	**M2**	**M3**	**M4**
640 × 480 1%	82 ± 29	37 ± 9	34 ± 23	17 ± 7
640 × 480 2%	119 ± 36	50 ± 10	48 ± 27	22 ± 8
640 × 480 3%	144 ± 392	58 ± 12	57 ± 29	24 ± 7
960 × 720 1%	68 ± 25	32 ± 10	27 ± 21	14 ± 7
960 × 720 2%	96 ± 32	41 ± 9	37 ± 24	18 ± 7
960 × 720 3%	118 ± 36	49 ± 10	45 ± 28	21 ± 8
1280 × 720 1%	58 ± 19	27 ± 8	20 ± 12	12 ± 6
1280 × 720 2%	82 ± 22	36 ± 7	29 ± 15	16 ± 7
1280 × 720 3%	102 ± 26	43 ± 8	37 ± 18	18 ± 7
1920 × 1080 1%	49 ± 19	24 ± 9	17 ± 15	10 ± 6
1920 × 1080 2%	67 ± 20	31 ± 8	23 ± 13	14 ± 6
1920 × 1080 3%	84 ± 23	37 ± 8	28 ± 15	16 ± 7

Tabela 6.5: Médias de FPS e desvios padrão para variadas resoluções de tela e limiares de *detail culling* por modelo.

do sistema operacional que chega no seu limite. Entretanto a distribuição de tarefas por *threads* ainda assim é crucial para a eficiência do processamento da gramática mesmo que não use todos os núcleos do processador.

Por fim, a Tabela 6.5 mostra o sumário de todos os FPS com seus desvios padrão em variadas resoluções de tela para a renderização síncrona. Podemos ver que utilizando baixas resoluções e/ou altos limiares de *detail culling* é possível conseguir taxas interativas para renderização síncrona até mesmo para o modelo M4.

6.5 Discussão

6.5.1 Consumo de Memória

A representação de modelos em MCAD *Shape grammar* obteve uma compactação de memória satisfatória, em todas as comparações a gramática foi mais compacta que as representações mínimas. Vale ressaltar que nas comparações utilizamos a versão contendo todas as regras de otimizações que descrevemos no Capítulo 5. Essa compactação poderia ser maior se não houvesse divisão de hierarquias entre os modelos, principalmente por separar objetos dos sistemas de uma unidade que vão ter padrões de objetos semelhantes. Para esse caso específico, uma abordagem interessante seria o compartilhamento de regras de produção inteira que gerasse um tipo de padrão específico, otimização que não implementamos nessa atual versão da conversão. Outro problema em juntar todos os objetos em um nível só para aumentar a compactação é que a derivação e interpretação seria forçada a ser feita toda sequencialmente, o que não permitiria que o modelo ficasse escalável para navegação em tempo real.

Utilizando *Shape matching* identificamos um grande volume de malhas duplicadas mesmo com a limitação de quantidade e ordenação de vértices que deve ser a mesma para o casamento entre malhas. Esse fato aliado com a grande quantidade de objetos paramétricos do modelo evidencia que pode existir uma quantidade limitada de padrões em projetos industriais que está implícita nos modelos, e que em geral não é explorada para fins de compactação por exemplo. Em termos de gramática, existe um vocabulário básico que contém as primitivas que são comuns no domínio de modelos CAD massivos (Ex: MCAD *Shape grammar*), assim como um vocabulário estendido deste de alto nível que descreve a configuração dos objetos para um projeto específico (Ex: A base PDMS que utilizamos nos testes). Em nossos experimentos, mesmo carregando um modelo na magnitude de M4, o sistema foi capaz de carregar o modelo inteiro na memória principal e da placa gráfica, dispensando esquemas *out-of-core* de geometrias como outras abordagens precisam utilizar para carregar modelos massivos [30, 27, 33].

Normalmente, os dados das transformações dos objetos são desprezíveis quando comparados ao consumo das malhas, onde seria o maior volume dos dados. Entretanto, uma vez que há uma grande redundância de tipos de geometrias, o volume de dados dessas transformações torna-se uma proporção relevante. A Tabela 6.3 mostra o consumo de memória entre malhas de polígonos e as regras produção que geram as transformações dos objetos, no qual podemos ver que as proporções já são diferentes do usual mesmo utilizando a gramática que é mais compacta que as outras descrições que mostramos na Figura 6.6. Para o modelo M1, as regras de produção consomem mais memória que as próprias malhas e em M4 as transformações correspondem a aproximadamente 30% do consumo de memória total do modelo. Por fim, a compactação dos dados com a representação da gramática vai contribuir não só na troca de dados como também no rápido carregamento do modelo em um

sistema de visualização.

6.5.2 Engine MCAD *Shape grammar*

A implementação da engine proposta nesse trabalho mostrou-se escalável em variados cenários. Em nossos testes utilizamos uma configuração de *hardware* não tão poderosa para os padrões atuais para aplicações de visualização em tempo real, e ainda assim obtemos resultados relevantes. Configurando o sistema com limiares de *detail culling* para resoluções de telas específicas, é possível atingir FPS interativo até mesmo navegando em um modelo como M4 que quando expandido pode ter mais de um bilhão de triângulos. Dessa forma, a performance da *engine* pode ser controlada conforme o quanto usuário aceita a qualidade da renderização para regiões pequenas ou distantes. Uma outra abordagem de configurar o sistema é manter um FPS alvo fixo, e alterar o limiar de *detail culling* em tempo de execução conforme a performance do renderizador. Alterar esse valor é instantâneo e possui efeito imediato na navegação conforme nossos testes realizados, mesmo para o caso do modelo ter sido convertido com divisão de LOD com um limiar conservador como 1%. Além do mais, esse limiar, por definição, só impacta em objetos pequenos em tela, as estruturas principais do modelo são preservadas como mostrado na Figura 6.4. Quando se amplia um detalhe da cena, mesmo na renderização assíncrona, os objetos que ora foram descartados por *detail culling* são renderizados rapidamente com o máximo de detalhe e melhor resolução possível quando forem superfícies paramétricas.

Um aspecto importante sobre o impacto do limiar de *detail culling* é em relação ao seu cálculo e sua precisão. Os limiares 1%, 2% e 3% correspondem a 10, 41 e 93 $pixels^2$ respectivamente, o que a princípio podem ser valores muito altos para descartar na imagem. Porém, os cálculos são baseados em esferas envolventes dos escopos dos objetos, que costumam ser bem menos ajustados que as OBB's, o que minimiza o impacto na prática quando os objetos são descartados, pois suas quantidades de *pixels* reais são bem menores que as de suas esferas envolventes. Para aumentar essa precisão, a solução poderia ser utilizar o cálculo baseado na OBB como em [24], porém em nossos experimentos esse cálculo em CPU não se mostrou eficiente. Embora o algoritmo seja eficiente em calcular a área da projeção da caixa em tela, a quantidade de objetos dos modelos massivos torna-o impraticável para ser realizado a cada quadro. Logo, é crucial que o cálculo de projeção na tela seja o mais rápido possível, mesmo que não tão precisa como a de esferas envolventes que utilizamos.

Em nossos testes, embora utilizemos modelos estáticos, estamos sempre gerando os *batches* de objetos novos a cada quadro com FPS interativo (que também tem desvantagens discutidas na Seção 6.5.3). Logo, nossa abordagem é moderadamente escalável para certos tipos de cenas dinâmicas também, uma vez que se consiga alterar a gramática original com eficiência. Nesse aspecto, o dinamismo na cena vai

ser mais eficiente se for incorporado diretamente na gramática, por exemplo, uma movimentação de objetos pode ser descrita na forma de regras de produção parametrizadas utilizando como variável o tempo que é atualizado a cada quadro. Essa ideia também pode ser aplicada na atualização da BVH, ou qualquer outra estrutura espacial modelada em gramática, cujos custos de atualizações são os maiores impedimentos de utilizar estruturas espaciais em cenas dinâmicas.

6.5.3 Limitações

A engine foi projetada para que seja eficiente renderizando primitivas especializadas em GPU e escalável com gramáticas derivadas e interpretadas *on-the-fly*. Entretanto, identificamos que temos um gargalo na transferência de dados, o que faz com que a renderização tenha eficiência reduzida em modelos de baixa escala quando comparadas à renderização direta. Isso se deve principalmente ao fato de que estamos sempre gerando *batches* novos, e descartando todos os dados do quadro passado. Uma abordagem seria tentar aproveitar parcialmente os dados dos *batches* anteriores e concatenar os novos dados. Entretanto, essa solução não é tão trivial, pelo fato da heterogeneidade dos tipos de objetos que podem ser incluídos no novo quadro e a remoção das instâncias antigas que podem fragmentar o *buffer* de objetos. A forma que enviamos os dados é uma das mais eficientes para serem desenhados, os atributos de um tipo de um objeto estão contíguos na memória e são renderizados com uma única chamada de desenho.

Na Figura 6.8, onde contém os caminhos da câmera para os testes, temos os instantes $t = 15s$, $t = 20s$, $t = 24s$ e $t = 55s$ como os pontos de mais baixa performance para os modelos M1, M2, M3 e M4 respectivamente, os quais são refletidos nos gráficos das Figuras 6.9 e 6.10. Nesses pontos específicos, os objetos estão ampliados e boa parte dos nós da BVH também estão visíveis, logo nenhuma de nossas regras de produção de aceleração são eficientes para esses casos, só conseguimos amenizar o gargalo aumentando o limiar de *detail culling*. Porém nesses pontos há uma oportunidade de aplicar *occlusion culling*, que não atacamos nessa atual implementação da *engine*. Logo, destacamos a necessidade de implementar alguma abordagem de *occlusion culling* genérico integrado na engine, ou criar regras que simulam o mesmo comportamento como discutido na Seção 4.8.2.

O *detail culling* foi um recurso que utilizamos extensamente para acelerar a renderização dos modelos e que possui um bom custo benefício entre qualidade de imagem e eficiência. Entretanto, existem casos em que seu limiar por mais baixo que seja pode gerar cenas com erro de imagem muito grande, a Figura 6.12 mostra um desses casos. Como apresentado na Seção 2.1.1 muitos objetos complexos CAD são formados pela composição de objetos menores, logo se um objeto é muito pulverizado em objetos pequenos, sua topologia vai ser comprometida. Uma forma de contornar utilizando a geração procedimental é substituindo o arranjo de objetos por um objeto mais simples de tal forma que preserve a topologia do objeto quando visualizada de

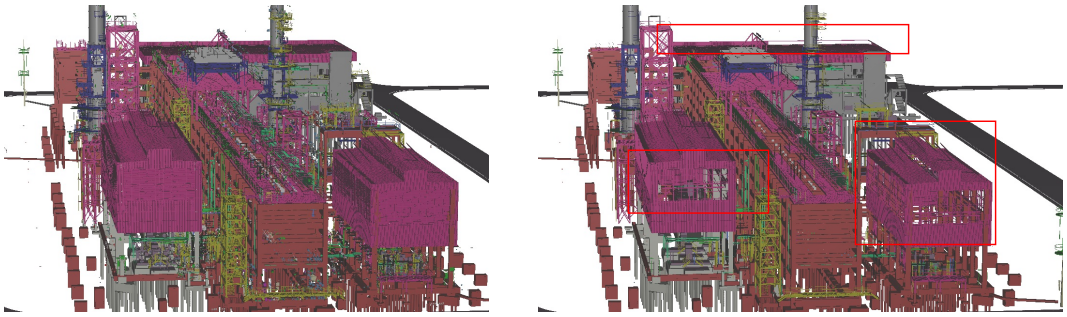

Figura 6.12: Estrutura perdida pelo descarte de *detail culling*. A esquerda a imagem com detalhe máximo. A direita detalhes perdidos destacados em vermelho.

longe, de forma análoga ao LOD discreto de malhas, porém como apresentado na Seção 4.2.2 o LOD é realizado a nível de objetos.

6.5.4 Análise de trabalhos relacionados

A Tabela 6.6 mostra o sumário do nosso trabalho e trabalhos relacionados com suas características e suas respectivas avaliações de performance. Escolhemos os trabalhos com os resultados mais relevantes e recentes, que foram publicados nos últimos 5 anos. Nas duas primeiras linhas colocamos nossas duas abordagens principais: renderização síncrona e assíncrona que têm características semelhantes aos trabalhos dos demais autores. Nas linhas dos trabalhos relacionados escolhemos a melhor performance para o modelo de maior magnitude reportados em seus textos originais. Não pudemos fazer uma comparação direta com todos os trabalhos, pois há variações nas origens dos modelos, além de configurações de máquinas diferentes. Os modelos Boeing 777 e o *Power Plant* são os modelos mais utilizados para *benchmark* no domínio, porém o modelo do Boeing 777 não é mais fornecido por seu fabricante[1] e o *Power Plant* já está obsoleto para as configurações de *hardware* atuais (128 MB e 12M de triângulos) [59].

Outro fator que vai variar entre os trabalhos são as estruturas de dados utilizadas por cada um. Steinberger et al. [12] gera uma "cidade infinita" expandindo um *Shape grammar* que gera prédios diretamente em GPU, assim como Marvie et al. [10] que além de prédios geram padrões de árvores. Em nossa própria abordagem, temos maior parte do modelo representado como objetos paramétricos, cuja resolução de malha vai influenciar na quantidade de triângulos da cena além das malhas de polígonos genéricas.

Em [13] propomos o MCAD *Shape grammar* e implementamos o renderizador de superfícies paramétricas e fizemos as primeiras avaliações renderizando no máximo o M2 com 30 FPS. Santos e Celes [31] utilizaram *Shape matching* e conseguiram

[1] `http://www.boeing.com/assets/pdf/commercial/airports/faqs/caddwgsprocedures.pdf`

renderizar o M3 com resolução de malha 16×16 com no máximo 11 FPS. As duas soluções são eficientes para renderizar até uma certa magnitude de modelos, porém não são escaláveis. Na *engine* que propormos nesse trabalho combinamos essas duas abordagens: a renderização com tessellation shaders e a instanciação de malhas duplicadas. Então, com processamento em CPU da gramática, pudemos navegar em modelos de maior ordem como o M4.

Peng et al. [30] e Xue et al. [33] ambos utilizaram esquemas *out-of-core* para renderizar o Boeing 777. Peng et al. implementou LOD diretamente na GPU e utilizou coerência quadro a quadro para reduzir o overhead entre CPU e GPU. Xue et al. aplicou uma "*voxelização*" no modelo para fazer consultas de visibilidades em CPU utilizando múltiplas *threads*. Os dados dos vértices foram comprimidos pela supressão das normais e quantizadas as posições dos vértices em 16 bits colocando-as relativas a uma AABB, a reconstrução é realizado na GPU onde são decodificados os vértices novamente no espaço do mundo no *vertex shader* e recalculadas as normais no *geometry shader*. Peng et al. atingiu aproximadamente 10 FPS no Boeing 777, com a limitação de ter a navegação prejudicada quando a coerência quadro a quadro é quebrada. Xue et al. obtiveram a performance de $30 \sim 45$ FPS no mesmo modelo. Considerando a versão assíncrona, no modelo M4 conseguimos renderizar a $18 \sim 99$ FPS, com a mesma estratégia de utilizar coerência utilizando o limiar de *detail culling* 1%. Na versão síncrona renderizamos $5 \sim 32$ para o mesmo limiar e até de $10 \sim 42$ para o limiar de 3%. Em ordem de triângulos, nossos modelos podem ser maiores que o Boeing 777, e ao ampliar os detalhes do modelo podemos ter uma visão dos objetos com alta resolução de triângulos; na abordagem de Xue et al., não é mencionada nenhuma estratégia de corrigir a perda na qualidade visual dos objetos pela compressão dos vértices das geometrias.

Steinberger et al. [12] implementaram uma abordagem baseada em *Shape grammar* para gerar prédios em tempo real e navegar em cidades massivas. O processo combina *frustum culling* e *occlusion culling* das caixas envolventes dos prédios balanceados por uma política de coerência quadro a quadro. No seu esquema de coerência, o algoritmo mantém prédios que estão obstruídos possivelmente temporariamente por um quantidade de quadros e carrega prédios que potencialmente ficarão visíveis nos próximos instantes. Assim como nossa abordagem assíncrona e a de Peng et al., movimentos bruscos da câmera ou navegação rápida na cena causam erros na renderização. O seu maior cenário consiste de 44K prédios que é uma cidade sintética que geram até 2B de triângulos com navegação a 20 FPS.

Marvie et al. [10] propuseram uma abordagem que utiliza *Shape grammars* para gerar geometrias diretamente na GPU. Semelhantemente a nossa abordagem, os autores utilizaram *tessellation shaders*, além de *geometry shader* para gerar alguns tipos de primitivas baseado nos parâmetros de símbolos terminais. Porém, essas primitivas são de mais baixo nível e criam objetos 3D pela composição de elementos de menores dimensões como vértices, linhas e *quads*. Quando os objetos são gerados, suas geometrias são armazenadas em *cache* para evitar gerar objetos estáticos por

quadro. Para LOD, a técnica gera impostores que são reutilizáveis pelas formas terminais também na GPU. O seu teste de maior complexidade, gerou uma cidade sintética com 116K prédios e 561K árvores renderizados a $7 \sim 12$ FPS. Segundo os autores a cena pode gerar mais 2TB de memória quando representada por geometria explícita, sendo compactada para 900MB em sua abordagem. Entretanto, os autores não reportaram a quantidade de polígonos gerados por sua cena.

A utilização de modelagem procedimental para gerar geometrias em GPU obteve bons resultados em gerar cenas massivas como mostrado em [10, 12, 39] para serem visualizadas em tempo real. Entretanto, esses trabalhos focaram-se mais no visual dos conjuntos de objetos que compõem a cena e na coerência entre eles. Em contraste com o nosso trabalho, nos preocupamos não só em manter taxas interativas e compactação de dados, mas preservando a representação funcional dos modelos para serem utilizáveis em projetos de engenharia. Temos um domínio mais complexo que somente prédios, árvores ou fractais que têm topologias bem-comportadas para gerar cenas massivas. Nossa renderização assíncrona renderiza um modelo real não aleatório de até 1,3B de triângulos com $18 \sim 99$ FPS com o mesmo problema de coerência da abordagem de Steinberger et al. Em relação a Marvie et al., não precisamos fazer cache de geometrias e nem de impostores, por termos primitivas mais alto nível e eficientes de serem renderizadas quadro a quadro, e obtemos taxas interativas para renderização síncrona como mostrado na Tabela 6.5.

	Abordagem	**Tipo de estrutura**	**Modelo base**	**Triângulos**	**FPS**
MCAD *Shape grammar engine* síncrona	Threads da engine sincronizadas	MCAD *Shape grammar*	M4	1,37B	$5 \sim 32$
MCAD *Shape grammar engine* assíncrona	Threads da engine não sincronizadas, navegação por coerência espacial	MCAD *Shape grammar*	M4	1,37B	$18 \sim 99$
Dos Santos et al. [13]	*Shape grammar*, renderização de superfícies paramétricas	MCAD *Shape grammar*	Equivalente ao M2	145M	22
Xu et al. [33]	Voxelização e esquema *out-of-core*	*Voxels*	Boeing 777	332M	$30 \sim 45$
Santos e Celes [31]	*Shape matching* e instanciação	Malha de polígonos	Equivalente ao M3	180M	10
Steinberger et al. [12]	*Shape grammar* em GPU, utiliza coerência quadro a quadro	*Shape grammar*	*Infinite City* (sintético)	2B	20
Peng et al. [30]	Coerência quadro a quadro e esquema *out-of-core*	Malha de polígonos	Boeing 777	332M	9
Marvie et al [10]	*Shape grammar* em GPU	*Shape grammar*	*Massive City* (sintético)	-	$7 \sim 12$

Tabela 6.6: Comparação com abordagens estado da arte em modelos CAD massivos e geração procedimental. Nas linhas correspondente ao nosso trabalho consideramos a resolução 960×720 com limiar de detail culling de 1%.

7
Considerações Finais

Nesse trabalho de tese fizemos uma pesquisa ampla da modelagem procedimental aplicada a modelos CAD massivos utilizando *Shape grammars*. Fizemos uma análise desde a conversão de modelos CAD industriais até a renderização dos objetos para visualização em tempo real. Não encontramos nenhum trabalho na literatura até então que utilizasse técnicas de modelagem procedimental em modelos reais pré-existentes como os que utilizamos, o que reforça a originalidade dessa pesquisa. Trabalhos de modelagem procedimental estão mais focados em gerar cenas aleatórias, porém coerentes e de forma rápida para um domínio específico. Em nossa abordagem exploramos as características de modelagem procedimental como padrões e repetições para compactação de modelos CAD massivos e renderização em tempo real. Concluímos que a abordagem procedimental tem muito potencial para ser explorada no domínio e como mostrado em nossos resultados a solução possui uma escalabilidade que atende os requisitos de modelos CAD massivos.

O nosso maior modelo de teste na sua forma original, sem considerar os objetos paramétricos, consome em forma de malhas de polígonos 8,17GB de memória de vídeo que é um consumo alto mesmo para as configurações de *hardware* atuais. Utilizando *Shape matching* para remoção de malhas duplicadas reduzimos esse volume de dados para 1,02GB que é um consumo mais aceitável. Dessa forma, concluímos que há um conjunto limitado de tipos de objetos que uma planta industrial pode conter. A redundância de formas geométricas define os padrões que vão caracterizar um domínio, cuja característica é explorada pela modelagem procedimental. Utilizando MCAD *Shape grammar* exploramos a repetição de tipos de objetos para compactação de dados e renderização eficiente.

Propomos uma engine que deriva e interpreta uma gramática *on-the-fly* para gerar cenas processando regras de produção que dividem a complexidade dos dados do modelo. O benefício dessa estratégia é que temos uma única estrutura de dados flexível que une descrição de modelos e estruturas de aceleração. Logo, na fase de modelagem temos a oportunidade de especificar LOD procedimentalmente que melhor gera um *feedback* visual para a navegação enquanto reduz o processamento de dados. Em nossa instância, sabendo que *detail culling* é executado durante o processamento do modelo para reduzir os dados de renderização, criamos regras de divisão de LOD de forma conservadora que antecipa o resultado do cálculo para grupos de objetos. Utilizando a gramática promovemos a aproximação da semântica dos objetos do modelo com a estrutura da modelagem, o que permite que

especifiquemos regras mais simples para otimizar o processamento. Por exemplo, conhecendo a estrutura hierárquica do modelo, criamos uma BVH que preserva a hierarquia, uma vez que objetos filhos de um mesmo nó possuem coerência espacial e obtemos resultados satisfatórios com essa divisão.

7.1 Contribuições

A seguir enumeramos nossas principais contribuições:

- **MCAD *Shape grammar***, uma gramática base expressiva suficiente para representar modelos CAD massivos industriais. A descrição em gramática especializa alguns tipos de objetos básicos e aplica restrições em malhas genéricas para que estejam contidas em seu escopo que são características importantes para a implementação da engine e compactação de dados;
- **Engine MCAD *Shape grammar***, uma engine que utiliza MCAD *Shape grammars* para navegação de modelos CAD em tempo real. Introduzimos uma derivação e interpretação simultânea que utiliza o contexto do escopo para produzir as variáveis *scope* e *lod* que são utilizadas para definir a derivação da gramática. Dessa forma, as estruturas de aceleração também estão descritas procedimentalmente o que contribui a manter uma descrição compacta. Uma vez que MCAD *Shape grammar* restringe o escopo para que represente a transformação e OBB de cada instância, podemos usá-las para fazer cálculos de *frustum culling* e realizar *detail culling* com corretude e eficiência. Por fim, as primitivas especializadas permitem que alguns tipos de objetos tenham um pipeline especial na engine para que tenha uma renderização mais otimizada;
- **Conversão de modelos**, baseado na definição da gramática MCAD *Shape grammar* e na forma que a engine a processa para visualização, criamos um pipeline de conversão de modelos pré-existentes para que gere suas respectivas gramáticas de forma compacta e escalável. As primitivas especializadas e a identificação de malhas redundantes reduzem consideravelmente o volume de malhas de triângulos, enquanto que o compartilhamento de escopos pela interpretação permite que compactemos as transformações dos objetos também, além da redução de instruções para o processo interpretação. Por fim, otimizamos as regras para acelerar o processamento com construção de BVH e divisão de LOD procedimentalmente.

A pesquisa dessa tese produziu o seguinte artigo:

[13] DOS SANTOS, Wallas HS; IVSON, Paulo; RAPOSO, Alberto Barbosa. **CAD *Shape Grammar*: Procedural generation for Massive CAD Model**. In: Graphics, Patterns and Images (SIBGRAPI), 2017 30th SIBGRAPI Conference on. IEEE, 2017. p. 31-38.

7.2 Trabalhos futuros

Como trabalhos futuros propomos um estudo mais aprofundado no uso da gramática na modelagem inicial de um projeto de planta industrial. Afinal, a modelagem procedimental teve sucesso em aplicações que geram cenas massivas de um domínio rapidamente pelo ajuste de parâmetros [11, 38]. Embora a gramática possa ser usada em variados níveis de abstração, seu uso direto como no formato texto pode não ser muito atraente para um usuário sem habilidades de programação, o que pode ser desafio aplicá-la na forma que é. Nesse caso acreditamos que uso de softwares com interfaces gráficas mais amigáveis possam ser um recurso que reduza essa dificuldade, como discutido em [60, 61]. Outra possibilidade é usar a gramática como uma camada de mais baixo nível para geometrias especificadas em arquivos IFC [62]. O IFC é o formato de arquivo padrão utilizado pela metodologia BIM para o domínio de modelos CAD industriais, que além de dados de geometrias vai conter outros dados de engenharia utilizados em seus casos de uso. Em sua especificação, as geometrias são descritas em alto nível, que por sua vez poderiam ser transformadas em regras de produção parametrizadas e decompondo-as em primitivas do MCAD *Shape grammar*. Uma vez transformada em regra de produção, podemos utilizar dos demais recursos que propomos como divisão de LOD em alto nível e renderização de primitivas especializadas que podem ser utilizadas por nossa engine proposta para gerar cenas com qualidade visual e fluidez.

Em nossa avaliação utilizamos modelos de um único projeto de um mesmo formato, porém estamos cientes que existem variações de padrões no domínio de modelos CAD massivos industriais. Por exemplo, modelos de plataformas de petróleo e o próprio Boeing 777 têm características semelhantes às dos modelos que trabalhamos além de outras que não cobrimos. Para esses outros modelos, consideramos que pode haver mais formas de explorar regras de produção como por exemplo realizar *occlusion culling* baseado na semântica da topologia como descrevemos na Seção 4.8.2. Além disso, há espaço para extensão de primitivas especializadas como por exemplo NURBS que vai ser mais comum em projetos que contêm estruturas com superfícies curvas ou outros objetos que podem ser descritos como superfícies implícitas. As extensões desses tipos de objetos são facilmente integráveis no pipeline da *engine*, assumindo que o tipo de objeto pode ser descrito por uma equação paramétrica, a mudança consistiria em enviar os seus parâmetros explícitos e somente alterar a etapa de *tessellation evaluation* onde é feita a deformação da malha gerada pelo *tessellation* na GPU.

Um dos nossos gargalos identificados nos testes foi a ausência de uma implementação de *occlusion culling*, que baixava a performance da *engine* quando havia uma grande parte dos objetos no *frustum* e o LOD ser alto. Temos duas sugestões de abordagens que não são exclusivas: integrar o algoritmo direto na *engine* ou oferecer recursos na gramática que explicitamente ordenasse a interpretação exe-

cutar consultas de visibilidade assim como fizemos com a variável *lod* e *scope*. Na primeira abordagem, uma forma de implementar um esquema de *occlusion culling* genérico é utilizar o *depth buffer* de um quadro e renderizar a caixa envolvente de um objeto e logo em seguida consultar quantos fragmentos foram gerados pela GPU para determinar se o objeto está ou quanto está obstruído. Porém, idealmente essa implementação só vai ser eficiente se considerar um grupo de objetos, para objetos individuais o algoritmo provavelmente não vai ficar escalável dado que as cenas podem gerar centenas de milhares de objetos por quadro. Então, a engine teria que ter uma forma de determinar em quais grupos de objetos se deve realizar o teste de oclusão. Na abordagem de integrar na gramática, essa responsabilidade fica por conta do usuário, porém pode ser uma solução mais eficiente. Em [12] a gramática para gerar prédios adiciona um dado na geometria chamado de *hull* que indica que aquela geometria deve ser usada para realizar cálculo de oclusão. Sugerimos duas formas de explicitar *occlusion culling* na gramática: criar uma variável *occluded* que possa ser usada na avaliação da regra de produção, ou generalizar a variável *scope* para que considere teste de *occlusion culling* além do *frustum culling*.

O outro gargalo de nossa engine proposta diz respeito a troca de dados entre CPU e GPU. Embora a forma que implementamos possa ser utilizada para certos tipos de cenas dinâmicas, podemos otimizar o processo para diminuir o *overhead* de transferência de dados entre CPU e GPU. Uma otimização relativamente simples do lado da interpretação é a utilização a coerência quadro a quadro identificando se houve mudanças de regras de produção avaliadas. Se não houve mudanças, logo o último *batch* enviado a engine continua válido, e a engine pode continuar a renderizá-lo mesmo na renderização síncrona. Um problema dessa abordagem é determinar se houve mudanças na visibilidade de objetos pelo *detail culling*, logo deve-se balancear também o quanto a mudança de um quadro para o outro influencia no resultado da visibilidade desses objetos. No lado da GPU, pode-se implementar um esquema de receber somente os dados novos da diferença entre quadros. Nesse caso, deve-se também identificar e remover dados que não estão sendo mais utilizados o que também gera o problema de fragmentação dos dados. Logo, pode ser necessário o uso de programação genérica em GPU para realizar a desfragmentação de dados diretamente na placa gráfica como feito em [30], evitando o *overhead* e aprimorando a eficiência do processo.

8
Referências bibliográficas

[1] HARDIN, B.; MCCOOL, D.. **BIM and construction management: proven tools, methods, and workflows**. John Wiley & Sons, 2015.

[2] SACKS, R.; TRECKMANN, M. ; ROZENFELD, O.. **Visualization of work flow to support lean construction**. Journal of Construction Engineering and Management, 135(12):1307–1315, 2009.

[3] DAWOOD, N.; SCOTT, D.; SRIPRASERT, E. ; MALLASI, Z.. **The virtual construction site (vircon) tools: An industrial evaluation**. Electronic Journal of Information Technology in Construction, 2005.

[4] FREIRE, V.; WANG, S. ; WAINER, G.. **Visualization in 3ds max for cell-devs models based on building information modeling**. In: PROCEEDINGS OF THE SYMPOSIUM ON SIMULATION FOR ARCHITECTURE & URBAN DESIGN, p. 9. Society for Computer Simulation International, 2013.

[5] HOU, L.; WANG, X. ; TRUIJENS, M.. **Using augmented reality to facilitate piping assembly: an experiment-based evaluation**. Journal of Computing in Civil Engineering, 29(1):05014007, 2013.

[6] BITTNER, J.; WONKA, P.. **Visibility in computer graphics**. Environment and Planning B: Planning and Design, 30(5):729–755, 2003.

[7] COHEN-OR, D.; CHRYSANTHOU, Y. L.; SILVA, C. T. ; DURAND, F.. **A survey of visibility for walkthrough applications**. IEEE Transactions on Visualization and Computer Graphics, 9(3):412–431, 2003.

[8] LUEBKE, D. P.. **Level of detail for 3D graphics**. Morgan Kaufmann, 2003.

[9] MÜLLER, P.; WONKA, P.; HAEGLER, S.; ULMER, A. ; VAN GOOL, L.. **Procedural modeling of buildings**. In: ACM TRANSACTIONS ON GRAPHICS (TOG), volumen 25, p. 614–623. ACM, 2006.

[10] MARVIE, J.-E.; BURON, C.; GAUTRON, P.; HIRTZLIN, P. ; SOURIMANT, G.. **Gpu shape grammars**. In: COMPUTER GRAPHICS FORUM, volumen 31, p. 2087–2095. Wiley Online Library, 2012.

[11] PRUSINKIEWICZ, P.; LINDENMAYER, A.. **The algorithmic beauty of plants**, volumen 1. Springer Science & Business Media, 2012.

[12] STEINBERGER, M.; KENZEL, M.; KAINZ, B.; WONKA, P. ; SCHMALSTIEG, D.. **On-the-fly generation and rendering of infinite cities on the gpu**. In: COMPUTER GRAPHICS FORUM, volumen 33, p. 105–114. Wiley Online Library, 2014.

[13] DOS SANTOS, W. H.; IVSON, P. ; RAPOSO, A. B.. **Cad shape grammar: Procedural generation for massive cad model**. In: GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), 2017 30TH SIBGRAPI CONFERENCE ON, p. 31–38. IEEE, 2017.

[14] IVSON, P.; NASCIMENTO, D.; CELES, W. ; BARBOSA, S. D.. **Cascade: a novel 4d visualization system for virtual construction planning**. IEEE transactions on visualization and computer graphics, 24(1):687–697, 2018.

[15] ZHOU, Y.; DING, L.; WANG, X.; TRUIJENS, M. ; LUO, H.. **Applicability of 4d modeling for resource allocation in mega liquefied natural gas plant construction**. Automation in construction, 50:50–63, 2015.

[16] MAHALINGAM, A.; KASHYAP, R. ; MAHAJAN, C.. **An evaluation of the applicability of 4d cad on construction projects**. Automation in Construction, 19(2):148–159, 2010.

[17] MESSNER, J. I.; YERRAPATHRUNI, S. C.; BARATTA, A. J. ; RILEY, D. R.. **Cost and schedule reduction of nuclear power plant construction using 4d cad and immersive display technologies**. In: COMPUTING IN CIVIL ENGINEERING (2002), p. 136–144. 2003.

[18] NICAŁ, A. K.; WODYŃSKI, W.. **Enhancing facility management through bim 6d**. Procedia engineering, 164:299–306, 2016.

[19] HECHT, J.. **Optical dreams, virtual reality**. Optics and Photonics News, 27(6):24–31, 2016.

[20] **Vulkan**. `https://www.khronos.org/vulkan/`.

[21] AKENINE-MÖLLER, T.; HAINES, E. ; HOFFMAN, N.. **Real-time rendering**. Crc Press, 2008.

[22] YOON, S.-E.; GOBBETTI, E.; KASIK, D. ; MANOCHA, D.. **Real-time massive model rendering**. Synthesis Lectures on Computer Graphics and Animation, 2(1):1–122, 2008.

[23] LUEBKE, D.; GEORGES, C.. **Portals and mirrors: Simple, fast evaluation of potentially visible sets**. In: PROCEEDINGS OF THE 1995 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, p. 105–ff. ACM, 1995.

[24] SCHMALSTIEG, D.; TOBLER, R. F.. **Fast projected area computation for three-dimensional bounding boxes**. Journal of graphics tools, 4(2):37–43, 1999.

[25] HOPPE, H.. **Progressive meshes**. In: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 99–108. ACM, 1996.

[26] SCHMALSTIEG, D.; TOBLER, R. F.. **Real-time bounding box area computation**. Vienna University of Technology, 1999.

[27] WALD, I.; DIETRICH, A. ; SLUSALLEK, P.. **An interactive out-of-core rendering framework for visualizing massively complex models**. In: ACM SIGGRAPH 2005 COURSES, p. 17. ACM, 2005.

[28] GOBBETTI, E.; MARTON, F.. **Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms**. In: ACM TRANSACTIONS ON GRAPHICS (TOG), volumen 24, p. 878–885. ACM, 2005.

[29] SOARES, L. P.; CORSEUIL, E. T.; RAPOSO, A. B.; GATTASS, M. ; SANTOS, I. H.. **Environ: Uma ferramenta de realidade virtual para projetos de engenharia**. CILAMCE-Iberian Latin American Congress on Comp Methods in Engineering, 2008.

[30] PENG, C.; CAO, Y.. **A gpu-based approach for massive model rendering with frame-to-frame coherence**. In: COMPUTER GRAPHICS FORUM, volumen 31, p. 393–402. Wiley Online Library, 2012.

[31] SANTOS, P. I. N.; CELES FILHO, W.. **Instanced rendering of massive cad models using shape matching**. In: GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), 2014 27TH SIBGRAPI CONFERENCE ON, p. 335–342. IEEE, 2014.

[32] VELTKAMP, R. C.; HAGEDOORN, M.. **State of the art in shape matching**. In: PRINCIPLES OF VISUAL INFORMATION RETRIEVAL, p. 87–119. Springer, 2001.

[33] XUE, J.; ZHAO, G. ; XIAO, W.. **Efficient gpu out-of-core visualization of large-scale cad models with voxel representations**. Advances in Engineering Software, 99:73–80, 2016.

[34] STINY, G.. **Introduction to shape and shape grammars**. Environment and planning B, 7(3):343–351, 1980.

[35] PARISH, Y. I.; MÜLLER, P.. **Procedural modeling of cities**. In: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, p. 301–308. ACM, 2001.

[36] BAO, F.; SCHWARZ, M. ; WONKA, P.. **Procedural facade variations from a single layout**. ACM Transactions on Graphics (TOG), 32(1):8, 2013.

[37] MERRELL, P.; SCHKUFZA, E. ; KOLTUN, V.. **Computer-generated residential building layouts**. In: ACM TRANSACTIONS ON GRAPHICS (TOG), volumen 29, p. 181. ACM, 2010.

[38] WONKA, P.; WIMMER, M.; SILLION, F. ; RIBARSKY, W.. **Instant architecture**, volumen 22. ACM, 2003.

[39] LIPP, M.; WONKA, P. ; WIMMER, M.. **Parallel generation of multiple l-systems**. Computers & Graphics, 34(5):585–593, 2010.

[40] STEINBERGER, M.; KENZEL, M.; KAINZ, B.; MÜLLER, J.; PETER, W. ; SCHMALSTIEG, D.. **Parallel generation of architecture on the gpu**. In: COMPUTER GRAPHICS FORUM, volumen 33, p. 73–82. Wiley Online Library, 2014.

[41] KRECKLAU, L.; PAVIC, D. ; KOBBELT, L.. **Generalized use of non-terminal symbols for procedural modeling**. In: COMPUTER GRAPHICS FORUM, volumen 29, p. 2291–2303. Wiley Online Library, 2010.

[42] BRUTZMAN, D.; DALY, L.. **X3D: extensible 3D graphics for Web authors**. Morgan Kaufmann, 2010.

[43] OSFIELD, R.; BURNS, D. ; OTHERS. **Open scene graph**. Library–OSG. http://www. openscenegraph. org, 2004.

[44] **Wavefront .obj**. `http://www.martinreddy.net/gfx/3d/OBJ.spec`.

[45] LIVNY, Y.; KOGAN, Z. ; EL-SANA, J.. **Seamless patches for gpu-based terrain rendering**. The Visual Computer, 25(3):197–208, 2009.

[46] TARIQ, S.; BAVOIL, L.. **Real time hair simulation and rendering on the gpu**. In: ACM SIGGRAPH 2008 TALKS, p. 37. ACM, 2008.

[47] GUTHE, M.; BALÁZS, A. ; KLEIN, R.. **Gpu-based trimming and tessellation of nurbs and t-spline surfaces**. In: ACM TRANSACTIONS ON GRAPHICS (TOG), volumen 24, p. 1016–1023. ACM, 2005.

[48] BAVOIL, L.; SAINZ, M.. **Screen space ambient occlusion**. NVIDIA developer information: http://developers. nvidia. com, 6, 2008.

[49] BARROSO, V. B. R.; CELES, W.. **Improved real-time shadow mapping for cad models**. In: COMPUTER GRAPHICS AND IMAGE PROCESSING, 2007. SIBGRAPI 2007. XX BRAZILIAN SYMPOSIUM ON, p. 139–146. IEEE, 2007.

[50] GLASSNER, A. S.. **An introduction to ray tracing**. Elsevier, 1989.

[51] **AVEVA**. `http://www.aveva.com/`.

[52] **RVM**. `https://knowledge.autodesk.com/support/navisworks-products/learn-explore/caas/`

CloudHelp/cloudhelp/2018/ENU/Navisworks/files/GUID-8C4487C7-7DD0-457D-BC08-B3628F33C173-htm.html.

[53] WOLD, S.; ESBENSEN, K. ; GELADI, P.. **Principal component analysis**. Chemometrics and intelligent laboratory systems, 2(1-3):37–52, 1987.

[54] CARLING, R.. **Matrix inversion**. In: GRAPHICS GEMS, p. 470–471. Academic Press Professional, Inc., 1990.

[55] MUSIALSKI, P.; WONKA, P.; ALIAGA, D. G.; WIMMER, M.; GOOL, L. V. ; PURGATHOFER, W.. **A survey of urban reconstruction**. In: COMPUTER GRAPHICS FORUM, volumen 32, p. 146–177. Wiley Online Library, 2013.

[56] **C++**. https://isocpp.org/.

[57] CREATION, G.-T.. **OpenGL mathematics**. URL http://glm. g-truc. net/0.9, 8, 2016.

[58] THE KHRONOS GROUP INC. **OpenGL**. https://www.opengl.org/.

[59] **Power plant model**. http://gamma.cs.unc.edu/Powerplant/.

[60] HOISL, F.. **Visual, interactive 3D spatial grammars in CAD for computational design synthesis**. PhD thesis, Citeseer, 2012.

[61] MCKAY, A.; CHASE, S.; SHEA, K. ; CHAU, H. H.. **Spatial grammar implementation: From theory to useable software**. AI EDAM, 26(2):143–159, 2012.

[62] **IFC**. http://www.buildingsmart-tech.org/specifications/ifc-overview.

[63] **OpenGL Tessellation**. https://www.khronos.org/opengl/wiki/Tessellation.

A
Superfícies paramétricas

Este anexo contém as equações paramétricas utilizadas para criar as superfícies das primitivas especializadas. Para cada figura das primitivas mostramos o padrão do *patch* gerado pelo *tessellation shader* com seus respectivos parâmetros *inner level* e *outer level* [63], onde a letra R corresponde a resolução da malha. As equações paramétricas estão em função das coordenadas do *patch* na forma (x, y), e os vértices de saída são representados por (v_x, v_y, v_z).

A.1
Cilindro

$$\begin{aligned} v_x &= \cos(2\pi y) \\ v_y &= \sin(2\pi y) \\ v_z &= x - 0.5 \end{aligned} \tag{A-1}$$

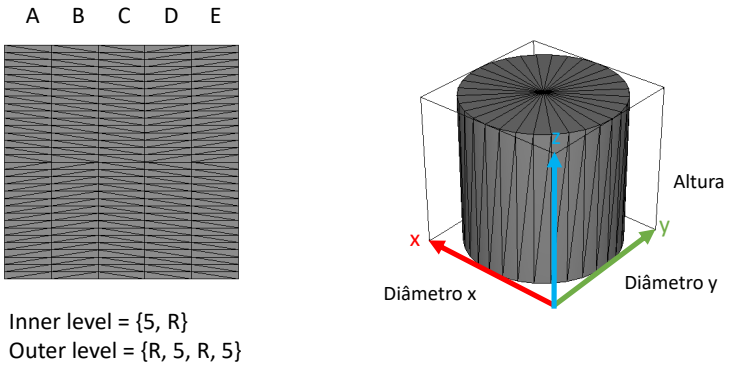

Figura A.1: Primitiva cilindro.

Nota: A corpo lateral do cilindro é a seção C. Para gerar as tampas utiliza-se as seções A e E, onde os vértices extremos são colocados no centro. As seções B e D são degeneradas para que as normais das outras partes fiquem consistentes.

A.2 Esfera

$$\begin{aligned} \theta &= 2\pi x \ , \ \phi = \pi y \\ v_x &= \frac{\cos(\theta)\sin(\phi)}{2} \\ v_y &= \frac{\sin(\theta)\sin(\phi)}{2} \\ v_z &= \frac{\cos(\phi)}{2} \end{aligned} \tag{A-2}$$

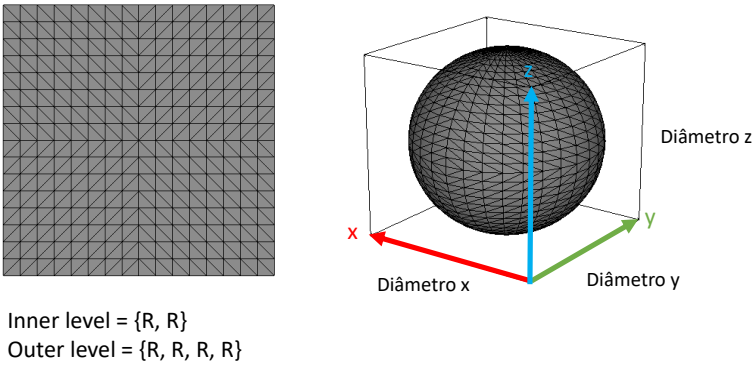

Figura A.2: Primitiva esfera.

A.3 Semiesfera

$$\begin{aligned} \theta &= \pi y \ , \ \phi = \tfrac{\pi x}{2} \\ v_x &= \frac{\cos(\theta)\sin(\phi)}{2} \\ v_y &= \frac{\sin(\theta)\cos(\phi)}{2} \\ v_z &= \sin(\phi) - \tfrac{1}{2} \end{aligned} \tag{A-3}$$

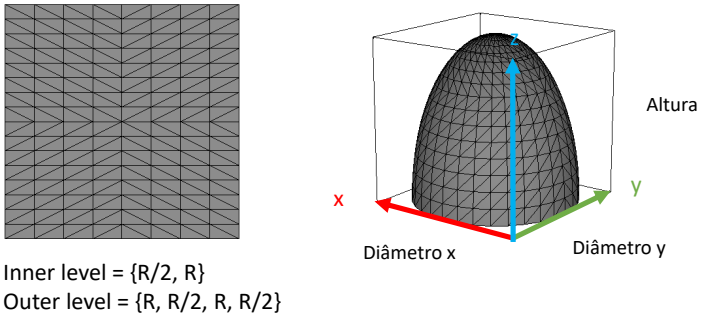

Figura A.3: Primitiva semiesfera.

A.4 Toro

Dado θ como ângulo de varredura, r_o raio exterior, r_i raio interior:

$$\begin{aligned} u &= x\ \theta\ ,\ v = 2\pi y \\ v_x &= r_i r_o \sin(v)\cos(u) \\ v_y &= r_i r_o \sin(v)\sin(u) \\ v_z &= r_i \cos(v) \end{aligned} \tag{A-4}$$

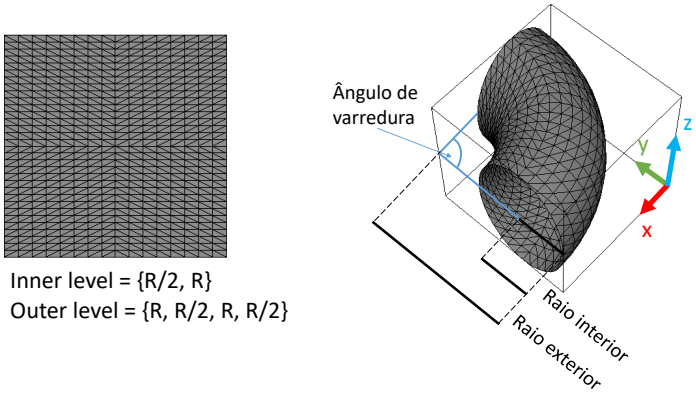

Figura A.4: Primitiva toro.

A.5 Cone

Dado r_b raio inferior, r_t raio superior, o_x deslocamento em x, o_y deslocamento em y:

$$\begin{aligned} u &= 2\pi y \\ v_x &= \frac{\cos(u)(r_b(1-x)+xr_t)}{o_x(x-1)} \\ v_y &= \frac{\sin(u)(r_b(1-x)+xr_t)}{o_y(x-1)} \\ v_z &= x - 0.5 \end{aligned} \tag{A-5}$$

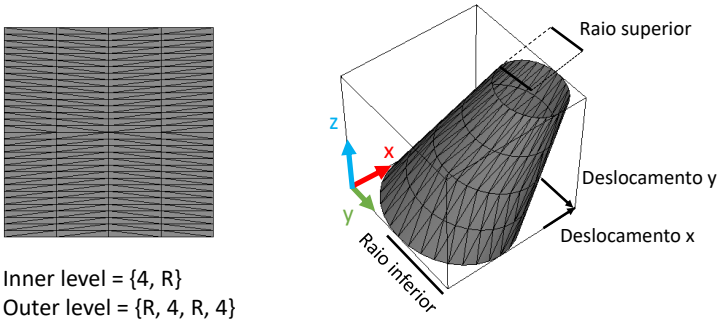

Figura A.5: Primitiva cone.

Nota 1: Para gerar a tampa do cone, pode ser feito de forma análoga ao cilindro.

Nota 2: As primitivas cone e toro devem ser normalizadas antes de aplicar a transformação do escopo. Isso pode ser feito pelo cálculo da caixa envolvente com seus respectivos parâmetros, então as dimensões são utilizadas para centralizar e normalizar os vértices.

B
Forma estendida Backus-Naur

A seguir a forma estendida Backus-Naur que verifica se uma gramática é aceita como MCAD *Shaper grammar*:

```
/*  grammar   */
mcadsg          ::=   production_rule (#x0A production_rule)

/*   expression  */
expression        ::= equation
equation          ::= binary_operation | term
term              ::= "(" ws* equation ws* ")" | "(" ws* number ws* ")" |
                       ws* number ws* | "(" ws* variable ws* ")" |
                       ws* variable ws*
binary_operation  ::= term ws* operator ws* term ws* ( operator equation)*

variable        ::= ([a-zA-Z] | "_")+ ([a-zA-Z] | [0-9])*
number          ::= "-"? ("0" | [1-9] [0-9]*) ("." [0-9]+)?
                    (("e" | "E") ( "-" | "+" )? ("0" | [1-9] [0-9]*))?
operator        ::= "+" | "-" | "*" | "/" | "^" | "%"
comp_operator   ::= (">" "=" ) | ("<" "=" ) | ("=" "=") | ">" | "=" | "<"

ws              ::= [#x20#x09]+ /* Whitespace:  Space | Tab */

/*  scope   */
scope_rule              ::= relative_translation | relative_rotation |
                            relative_scale | absolute_translation |
                            absolute_rotation | absolute_scale | "[" | "]"
scope_arguments         ::= "(" expression "," expression "," expression ")"
relative_translation    ::= "T" scope_arguments
relative_rotation       ::= "R" scope_arguments
relative_scale          ::= "S" scope_arguments
absolute_translation    ::= "M" scope_arguments
absolute_rotation       ::= "G" scope_arguments
absolute_scale          ::= "E" scope_arguments
```

```
/* production rule */

production_rule     ::= ws* predecessor ws* "-" ">" ws* sucessor ws*
                        (#x0A production_rule)*

predecessor         ::= variable  parameters? conditions?
parameters          ::= "(" ws* variable ws* ("," ws* variable ws*)* ")"
conditions          ::= "[" ws* variable ws* comp_operator expression ws* "]"

parameterized_rule  ::= variable "(" expression ("," expression)* ")"
sucessor            ::= (repeat | split | scope_rule | instance_rule |
                         parameterized_rule | variable) ws* sucessor*


/* instance */

instance_rule   ::= cube | cylinder | sphere | dish | torus | cone | mesh
cube            ::= "I" "(" #x22 "c" "u" "b" "e" #x22 ")"
cylinder        ::= "I" "(" #x22 "c" "y" "l" "i" "n" "d" "e" "r" #x22 ")"
sphere          ::= "I" "(" #x22 "s" "p" "h" "e" "r" "e" #x22 ")"
dish            ::= "I" "(" #x22 "d" "i" "s" "h" #x22 ")"
torus           ::= "I" "(" #x22 "t" "o" "r" "u" "s" ws+
                    number ws+ number ws+ number #x22 ")"
cone            ::= "I" "(" #x22 "c" "o" "n" "e" ws+
                    number ws+ number ws+ number ws+ number #x22 ")"
mesh            ::= "I" "(" #x22 instance_name+ #x22 ")"
instance_name   ::= [a-zA-z] | [0-9] | "_" | " " | "."

/* Split operations */

repeat     ::=  "R" "e" "p" "e" "a" "t" "(" ws*
                #x22 axis? axis? axis? #x22 ws* "," ws* number ws*")"
                "{" ws* variable (ws+ variable)* ws* "}"
split      ::=  "S" "p" "l" "i" "t" "(" ws*
                #x22 axis? axis? axis? #x22 ws* "," ws* number "r"?
                (ws* "," ws* number "r"?)* ws*")"
                "{" ws* variable (ws+ variable)* ws* "}"
axis       ::=  "X" | "Y" | "Z"
```