

**UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**EXPLORAÇÃO DA LINGUAGEM RUST PARA
O DESENVOLVIMENTO DE UM *PATH*
TRACER PARALELO**

TRABALHO DE GRADUAÇÃO

Yuri Kunde Schlesner

Santa Maria, RS, Brasil

2014

EXPLORAÇÃO DA LINGUAGEM RUST PARA O DESENVOLVIMENTO DE UM *PATH TRACER* PARALELO

Yuri Kunde Schlesner

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da
Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a
obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Prof^a. Dr^a. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2014

**Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

**EXPLORAÇÃO DA LINGUAGEM RUST PARA O DESENVOLVIMENTO
DE UM *PATH TRACER* PARALELO**

elaborado por
Yuri Kunde Schlesner

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Dr^a.
(Presidente/Orientadora)

Iara Augustin, Prof^a. Dr^a. (UFSM)

João Vicente Ferreira Lima, Prof. Dr. (UFSM)

Santa Maria, 20 de Outubro de 2014.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

EXPLORAÇÃO DA LINGUAGEM RUST PARA O DESENVOLVIMENTO DE UM *PATH TRACER* PARALELO

AUTOR: YURI KUNDE SCHLESNER

ORIENTADORA: ANDREA SCHWERTNER CHARÃO

Local da Defesa e Data: Santa Maria, 20 de Outubro de 2014.

A Computação Gráfica, e especificamente a geração de imagens por computador, é uma área com inúmeras aplicações. No entanto, para a geração de imagens realísticas, é essencial fazer uso eficiente do poder computacional disponível, pois é um processo que requer a realização de um grande número de cálculos. Tradicionalmente foram utilizadas linguagens como C++, que permitem esta eficiência, para escrever estes tipos de sistemas. Rust, uma nova linguagem de sistemas, tem como um dos seus objetivos servir este nicho de aplicação. Este trabalho tem como objetivo re-implementar um renderizador fotorrealista utilizando Rust, afim de avaliar se a linguagem realmente atende as necessidades deste tipo de aplicação.

Palavras-chave: Computação Gráfica. Linguagens de Programação. Programação Paralela. Rust. Path Tracing.

SUMÁRIO

LISTA DE FIGURAS	6
LISTA DE TABELAS	7
1 INTRODUÇÃO	8
1.1 Objetivos	9
1.1.1 Objetivo Geral	9
1.1.2 Passos de metodologia	9
1.2 Justificativa	10
2 FUNDAMENTOS E REVISÃO DE LITERATURA	11
2.1 Rust	11
2.1.1 <i>Lifetimes</i>	12
2.1.2 Prevenção de <i>Aliasing</i>	13
2.1.3 <i>Traits</i>	13
2.1.4 <i>unsafe</i>	15
2.2 Path Tracing	16
2.3 SmallVCM	18
2.3.1 Paralelização	19
3 DESENVOLVIMENTO	20
3.1 Ambiente de Desenvolvimento e Organização de Projeto	20
3.1.1 Rust	20
3.1.2 C++	21
3.2 Metodologia da <i>Port</i>	21
3.3 Mudanças e Dificuldades	22
3.3.1 Herança	22
3.3.2 Sobrecarga de Funções	23
3.3.3 Sobrecarga de Operadores e Conversões Automáticas	23
3.3.4 Instabilidade	26
3.3.5 <i>Enums</i>	26
3.3.6 Ponteiros Nulos	27
3.3.7 Biblioteca padrão	28
4 RESULTADOS	29
4.1 Testes de Desempenho	29
5 CONCLUSÃO	31
REFERÊNCIAS	32

LISTA DE FIGURAS

2.1	Exemplos de uso de referências e restrições de <i>aliasing</i>	14
2.2	Um exemplo de uma imagem gerada utilizando <i>path tracing</i>	17
2.3	Uma imagem gerada pelo SmallVCM.	18
3.1	Classe abstrata <code>AbstractRenderer</code> e sua conversão para uma <i>struct</i> e <i>trait</i>	24
3.2	Definição e uso de uma macro e sua expansão	25
3.3	Comparação entre <i>enums</i> em C++ e Rust	27

LISTA DE TABELAS

4.1	Tempo de execução do algoritmo <code>EyeLight</code> em diferentes implementações e compiladores.	30
-----	--	----

1 INTRODUÇÃO

A *Computação Gráfica* é a área da Ciência da Computação que estuda tópicos relacionados à criação, análise e manipulação de imagens e conceitos relacionados. Dentre estas, a síntese (ou renderização) de imagens é onde uma imagem é criada de forma computacional, a partir de um modelo matemático e frequentemente buscando o fotorrealismo. Tem uma vasta quantidade de aplicações práticas: usada na engenharia, durante o projeto de máquinas ou construções; na arquitetura, para a visualização de espaços; para entretenimento, em efeitos especiais de filmes ou em jogos 3D e muitas outras.

Como a geração de imagens fotorrealistas envolve essencialmente uma simulação completa da física da luz, um processo proibitivamente lento e complexo, são utilizadas simplificações e modelos. No passado, devido à limitada capacidade computacional disponível, eram utilizadas aproximações grosseiras que, embora produzissem imagens atrativas, não eram muito realistas, especialmente no quesito da aparência das superfícies e de suas interações com a luz. Com o aumento do poder computacional disponível, vem sendo usados modelos mais fiéis à realidade e que produzem imagens mais convincentes, algumas vezes indistinguíveis de uma fotografia real.

Path tracing é um método de renderização que assume que a luz se comporta como uma partícula e calcula uma imagem traçando uma série de raios pelos caminhos através quais a luz viajaria quando refletida através de uma cena. Atualmente, é um dos algoritmos mais usados quando são demandadas imagens com um grau de realismo extremamente alto devido à habilidade de simular o comportamento da luz com relativa precisão. (PHARR; HUMPHREYS, 2010)

No entanto, este realismo vem ao custo de muito poder de processamento. Mesmo com o avanço tecnológico de CPUs, a renderização de imagens continua sendo uma tarefa árdua para processadores. Sistemas de renderização profissionais são quase exclusivamente escritos em C++ e não em linguagens de mais alto nível, devido às penalidades de desempenho que impõem. Sistemas mais recentes chegam a fazer o uso de GPUs para acelerar a imensa quantidade de cálculos necessária. Tendo em vista a baixa expressividade de C++ comparada a estas outras linguagens, torna-se interessante explorar alternativas que permitam desenvolvimento mais fácil sem sacrificar o desempenho requerido.

A linguagem de programação Rust, (Mozilla Research, 2010) um projeto de pesquisa da Mozilla Research, tem como seu objetivo ser uma união entre linguagens de programação

de sistemas e as tidas como “linguagens de alto-nível”, focando simultaneamente em alto-desempenho, segurança e expressividade. Ela atinge isso usando um modelo tradicional de compilação prévia (*ahead of time*), gerando código nativo que executa sem utilizar uma máquina virtual, e um sistema de tipos que permite a verificação automática dos usos de ponteiros durante a compilação, eliminando a possibilidade de acontecerem erros de memória sem introduzir penalidades excessivas de desempenho ou consumo de memória. Ao mesmo tempo, integra conceitos mais recentes de linguagens de programação que aumentam sua expressividade e capacidade de facilmente descrever programas complexos.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é portar um *path tracer* para a linguagem Rust e, através deste processo, realizar uma comparação quantitativa e qualitativa entre as versões escritas nesta linguagem e C++, nos aspectos de desempenho e organização de código, respectivamente. Como base será utilizado o SmallVCM (DAVIDOVIČ, 2012), um *path tracer* de pesquisa escrito em C++, escolhido por implementar uma variedade de algoritmos diferentes de *path tracing*, ser paralelizado utilizando *multi-threading* e por ser relativamente compacto, consistindo de aproximadamente 5000 linhas de código.

1.1.2 Passos de metodologia

- Estudar o código original do SmallVCM.
- Re-escrever uma parte do SmallVCM utilizando Rust, para a realização de testes.
- Realizar uma comparação de desempenho e expressividade de linguagem entre as duas versões, atentando a possíveis pontos onde a Rust pode ser melhorada.
- Paralelizar a nova versão do renderizador, fazendo uso das funcionalidades de Rust.
- Portar o restante do SmallVCM, afim de que as duas versões tenham funcionalidades equivalentes.

1.2 Justificativa

Rust é uma linguagem relativamente nova e, embora aplicações gráficas de alto desempenho sejam um dos seus públicos alvo, ainda não existe uma quantidade significativa de programas deste tipo que valide a linguagem para este propósito. A experiência e resultados adquiridos durante a realização deste trabalho podem ajudar a guiar o desenvolvimento da linguagem para atingir este fim.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

Neste capítulo é dada uma descrição dos conceitos teóricos das áreas de Linguagens de Programação e de Computação Gráfica nas quais o trabalho e as ferramentas nele utilizadas, a linguagem de programação Rust e o algoritmo de *path tracing*, se baseiam.

2.1 Rust

Rust (Mozilla Research, 2010) é uma linguagem de programação de propósito geral, oficialmente patrocinada pela Mozilla Research mas desenvolvida também pela comunidade. Ela busca atender as necessidades de programadores que atualmente utilizariam linguagens como C ou C++, por questões de desempenho ou de controle sobre o hardware. Uma de suas características mais distintivas é o modelo de referências que utiliza, baseado em regiões (GROSSMAN et al., 2002), que automaticamente gerencia a vida de alocações de memória, evitando com que sejam feitos erros neste gerenciamento, ou que memória inválida seja acessada pelo programa, sem necessitar o uso de um *garbage collector*. A linguagem, originalmente projetada por Graydon Hoare como um projeto pessoal, foi adotada pela Mozilla Research afim de servir como linguagem de implementação do projeto Servo (Mozilla Research, 2012), um browser experimental de próxima geração, mas atualmente já cresceu além deste objetivo para se tornar um projeto maior.

Além de melhorias no gerenciamento de memória, a linguagem busca também trazer funcionalidades tradicionalmente oferecidas em linguagens funcionais a programação procedural, trazendo uma forte inspiração de linguagens como ML (MILNER, 1997). Assim, possui funcionalidades como *pattern matching*, e *algebraic data types* como tuplas e *sum types* (Também conhecidos como tipos variantes ou uniões discriminadas.) A linguagem não implementa o modelo de herança tradicional de programação orientada a objetos, oferecendo em troca o conceito de *traits*.

O compilador oficial da linguagem, o rustc, é implementado na própria linguagem e utiliza a infraestrutura do projeto LLVM (LATTNER; ADVE, 2004), permitindo que usufrua das capacidades de otimização e geração de código do projeto para alcançar uma variedade de plataformas. O compilador também tem suporte ao carregamento de *plugins* durante o processo de análise, o que expõe poderosas capacidades de meta-programação aos usuários. Alguns exemplos de projetos que utilizam estas capacidades são o RustGPU (HOLK et al., 2013), que

permite a compilação de código Rust para execução paralela em GPUs, e Zinc (POUZANOV, 2014), um *framework* para desenvolvimento de aplicações embarcadas em microprocessadores.

2.1.1 *Lifetimes*

Para criar uma linguagem segura, eficiente e sem *runtime*, é necessário assegurar-se que todos os ponteiros, ou referências, no programa sejam válidas quando são acessadas. Para que isto seja possível sem incidir um custo na execução, esta é uma tarefa que precisa ser feita durante a compilação do programa, e não durante sua execução, como é feito em muitas linguagens onde todos os acessos de ponteiro são testados por validade antes de serem executados ou onde esta tarefa é deixada a cargo do programador, resultando em eventuais erros.

Rust modela esta restrição fazendo uma análise do fluxo de valores e referências na função, que tem sua vida delimitada por escopos léxicos no programa. É permitido que uma variável seja usada em escopos dentro do qual ela foi definida, e quaisquer referências criadas a esta variável tem seu tipo associado ao escopo, de forma a proibir que estas escapem para um escopo superior (onde elas não seriam mais necessariamente válidas, pois a variável teria sido destruída no fim do seu respectivo escopo.) A região na qual uma variável é válida e referências a ela podem existir é chamada de sua *lifetime*.

Para suportar a mobilidade de valores para escopos mais externos, variáveis também podem ser movidas. Mover uma variável (para outra variável, para dentro de uma chamada de função ou retornando-a para fora de uma função) invalida a variável original que continha o objeto. O compilador proíbe acessos a esta variável após ela ter sido movida, pois seus conteúdos não são mais válidos. Por esta mesma razão, é proibido mover uma variável para a qual existam referências, que seriam invalidadas se isto acontecesse. Por isso, o ato de criar uma referência a partir de uma variável é chamado *borrow* (“emprestar”): o direito de vida da variável é compartilhado com a referência, enquanto esta existir.

A mesma limitação de mover para fora de uma variável emprestada também limita modificações à variável original. Isto evita uma classe de problemas causados pela modificação de um objeto enquanto existam referências à objetos contidos nele. Utilizando-se um vetor redimensionável, por exemplo, poderia ocorrer uma situação em que é criada uma referência a um elemento dentro do vetor. Se forem então permitidas modificações ao vetor, este poderia ser redimensionado de forma a invalidar a referência.

Existem situações em que esta restrição de referências baseada no escopo léxico é limitada

demais. Nestes casos é feito o uso de *variáveis de lifetime*, que permitem que referências sejam retornadas de funções ou armazenadas em estruturas. Uma variável de *lifetime* nomeada é adicionado a uma função, e pode ser usada para especificar que certas referências criadas dentro da função são válidas dentro de um escopo maior que a chamada, correspondendo invés a um escopo herdado da função que a chamou. (Rust Project Developers, 2014a)

Exemplos

2.1.2 Prevenção de *Aliasing*

Para facilitar a análise do comportamento de um programa pelo usuário, Rust também impõe restrições quanto a mutabilidade de referências a objetos. Referências *únicas* ou *mutáveis* permitem a modificação do objeto a que apontam, mas restringem qualquer acesso a ele sem ser pela referência enquanto estiver emprestado. Referências *compartilhadas* ou *imutáveis* restringem a modificação do objeto para a qual apontam, mas permitem a existência de várias delas apontando para o mesmo. Isto garante que, se uma função receber uma referência compartilhada, o objeto a qual ela aponta nunca será modificado durante a duração da função, não precisando o programador ou o compilador preocupar-se com efeitos colaterais de operações em outros objetos. Da mesma forma, se uma função receber uma referência única, poderá modificá-la sem se preocupar que isto irá interferir com outros objetos ou referências presentes dentro da função. Alguns exemplos destas situações são mostrados na figura 2.1. (Rust Project Developers, 2014a)

Esta garantia de que valores só podem ser modificados através de um único ponto por vez também automaticamente assegura que não existam erros de sincronização em programas concorrentes. Dados compartilhados por vários *threads* serão acessados exclusivamente através de referências compartilhadas, com a garantia de que não serão modificados, e assim não precisam de exclusão mútua para evitar condições de corrida. Por outro lado, se um *thread* possuir uma referência única, sabe-se que este dado não pode ser lido em nenhum outro lugar, também evitando a necessidade de sincronismo durante modificações.

2.1.3 *Traits*

O projeto de Rust considera importante o suporte a programação genérica sobre tipos polimórficos suporta o conceito de herança de classes. No entanto, ainda é essencial suportar o uso polimórfico de objetos. Este suporte é fornecido através do conceito de *traits*. *traits* são similares aos conceitos de *interfaces* ou *type-classes* (WADLER; BLOTT, 1989): um tipo

```
fn main() {
    let mut n = 52u32;

    let a = &n;
    let b = &n; // OK - Duas referencias somente-leitura

    println!("{}", *a, *b);
}
```

(a)

```
fn main() {
    let mut n = 52u32;

    let a = &n;
    let b = &n; // OK - Duas referências somente-leitura

    // ERRO - Não pode modificar referência compartilhada
    *a = 24;

    println!("{}", *a, *b);
}
```

```
<anon>:8:5: 8:12 error: cannot assign to
      immutable dereference of `&`-pointer `*a`
<anon>:8      *a = 24;
              ^~~~~~
```

(b)

```
fn main() {
    let mut n = 52u32;

    let a = &mut n; // OK - Uma referência mutável

    *a = 24; // OK

    println!("{}", *a);
}
```

(c)

```
fn main() {
    let mut n = 52u32;

    let a = &mut n;
    let b = &n; // ERRO - n já tem uma referência mutável

    println!("{}", *a, *b);
}
```

```
<anon>:5:14: 5:15 error: cannot borrow `n`
      as immutable because it is also borrowed
      as mutable
<anon>:5      let b = &n;
              ^
<anon>:4:18: 4:19 note: previous borrow of
      `n` occurs here; the mutable borrow
      prevents subsequent moves, borrows, or
      modification of `n` until the borrow ends
<anon>:4      let a = &mut n;
              ^
```

(d)

Figura 2.1: Exemplos de uso de referências e restrições de *aliasing*.

apresentando uma interface que consiste apenas de declarações de funções, sem campos de dados ou implementação, e que forma um contrato especificando a interação entre dois componentes do código. Estas funções devem ser posteriormente implementadas por um tipos concretos, de acordo com o contrato estabelecido, permitindo que estes tipos sejam usados de forma genérica através da interface. Um dado tipo pode implementar mais de uma interface, permitindo expor vários conjuntos de funcionalidade distintos. (Rust Project Developers, 2014b) **exemplo**

Uma vantagem de *traits* sobre o modelo de herança tradicional é que se evita a ambiguidade presente em casos onde duas superclasses em de um tipo herdam de um mesmo tipo compartilhado. Isto cria uma ambiguidade onde cada classe pode conter dentro de si uma instância separada dos campos das classes pai ou onde ambas compartilham os mesmos campos. Esta situação é chamada de “problema diamante”, devido a estrutura do grafo de herança resultante. Por esta razão, muitas linguagens restringem seu suporte à herança simples, permitindo que classes herdem de somente uma superclasse. (SCHÄRLI et al., 2003) **diagrama**

Outra vantagem é a possibilidade de se estender tipos através da definição posterior de um novo *trait*. Como a implementação de um *trait* para certo tipo é separada da definição do tipo em si, pode-se fazer um tipo pré-existente suportar novas interfaces. **exemplo junto com o 1o?**

2.1.4 `unsafe`

Caso as limitações anteriormente descritas que são impostas no gerenciamento de memória e em referências mostrem-se muito restritivas durante a implementação de alguma estrutura de dados, algoritmo ou interface com outras bibliotecas ou APIs externas, existe a opção de fazer o uso de um bloco `unsafe`. Blocos `unsafe` não alteram a semântica da linguagem, mas relaxam restrições do verificador estático acerca de operações que não garantem a correteude do programa. Foram criados para expandir o campo de aplicação da linguagem, pois permitem que código Rust faça chamadas a funções definidas em outras linguagens (como funções de bibliotecas C ou do SO) e a conversão de referências para ponteiros não restritos. Cabe então ao programador verificar que as operações contidas nestes blocos não quebrem as invariantes assumidas pelo sistema de tipos. A intenção é que estes blocos sejam utilizados para implementar abstrações seguras por cima das operações necessárias, sendo facilmente identificados para fins de verificação de seu correto funcionamento. (Rust Project Developers, 2014c)

A biblioteca da padrão, implementada completamente em Rust, inclui uma multitude de tipos que utilizam esta estratégia para fornecer contrapartidas checadas em *runtime* das

invariantes do sistema de tipos, como ponteiros com contagem de referências (tipo $\text{Rc}<\text{T}>$), uma variedade de estruturas de dados e estruturas de concorrência, como o tipo $\text{Mutex}<\text{T}>$, que impede que um valor seja acessado sem antes adquirir uma trava.

2.2 *Path Tracing*

Path tracing faz parte de uma família de algoritmos comumente denominados algoritmos de *ray tracing*. Embora também utilizados na física e nas engenharias, no contexto deste trabalho são algoritmos que tem como finalidade a produção de imagens que retratam cenas tridimensionais.

Todos os algoritmos desta família se baseiam na ideia fundamental de simular o comportamento da luz traçando raios que saem da câmera virtual em direção à cena. Isto é o contrário do que ocorre na vida real, onde a luz é emitida de uma fonte e viaja pelo espaço até chegar ao observador, mas não afeta negativamente o resultado final e torna o algoritmo computacionalmente viável, pois assegura que todo o raio traçado é um que eventualmente chegaria no observador: a maioria da luz numa cena não chega até o observador, que subtende um espaço relativamente pequeno nela. (PHARR; HUMPHREYS, 2010)

Os primeiros algoritmos deste tipo a serem usados simplesmente traçavam um raio por pixel da imagem, encontrando a intersecção deste raio com a cena e calculando sua aparência de acordo com algum modelo básico de iluminação. Desta forma, não eram reproduzidas sombras nem superfícies refletivas, como espelhos ou objetos metálicos. Estes tipo de algoritmos vieram a ser chamado de algoritmos de *ray casting*.

WHITTED (1980) propôs um novo método que soluciona estes problemas. Além do primeiro raio partindo da câmera, são também traçados raios que vão do ponto sendo iluminado até cada uma das fontes luminosas, permitindo que cada luz só seja adicionada se não estiver obstruída por outro objeto e assim permitindo a renderização de sombras. Em superfícies refletivas, outro raio é traçado na direção do reflexo, o qual é utilizado para calcular a luminosidade naquela direção, da mesma maneira que o raio inicial. Assim, este algoritmo implementa *ray tracing recursivo*.

COOK; PORTER; CARPENTER (1984) aprimoraram o algoritmo de WHITTED para que suporte uma variedade de efeitos adicionais como superfícies foscas e translúcidas, sombras com penumbras realísticas, profundidade de campo e borrão de movimento. Todos estes efeitos são realizados através do mesmo método de tirar várias amostras em cada ponto da imagem,

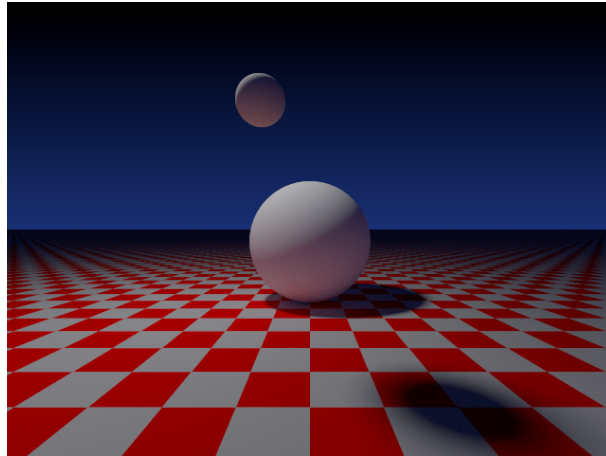


Figura 2.2: Um exemplo de uma imagem gerada utilizando *path tracing*. Note como a luz que atinge o plano xadrez é refletida de volta para iluminar a esfera, um fenômeno conhecido como *iluminação indireta* e que é corretamente simulado pelo algoritmo.

introduzindo uma variação nas direções ou posições traçadas em cada amostra. Embora não tenha o embasamento matemático, esta é a mesma ideia básica utilizada posteriormente em algoritmos que utilizam integração Monte Carlo.

KAJIYA (1986) introduz a *equação de renderização*, que descreve a interação da luz com as superfícies, modelando também a reflexão de luz em superfícies completamente foscas (ver Figura 2.2.) Esta serve como uma importante fundação teórica que é usada como base para o cálculo da imagem ou para o desenvolvimento de aproximações. No mesmo artigo é introduzida a técnica de *path tracing*, que difere das anteriores desenvolvidas por COOK; PORTER; CARPENTER e WHITTED por observar que os raios mais impactantes na aparência final da imagem são os de baixa profundidade, e assim traçando apenas um raio recursivo por amostra, evitando o crescimento exponencial do número de raios traçados que ocorre com as outras técnicas.

Embora capaz de produzir imagens extremamente realísticas, *path tracing* pode requerer uma quantidade impraticável de amostras para renderizar satisfatoriamente certos tipos de cenas onde não exista uma linha de visão direta entre as superfícies e as fontes de luz. Nestas cenas, a maioria da iluminação se dá através de caminhos indiretos ou através de superfícies refratantes que projetam padrões de luz complicados em outras superfícies (conhecidos como *caustics*.) Para contornar estes problemas foram desenvolvidas inúmeras extensões ao algoritmo de *path tracing*. Dentre elas se destaca *bidirectional path tracing*, introduzido por LAFORTUNE; WILLEMS (1993). Este algoritmo foi depois reformulado em (VEACH, 1997), onde também foi introduzida a técnica de *multiple importance sampling* e o algoritmo *Metropolis light transport*.

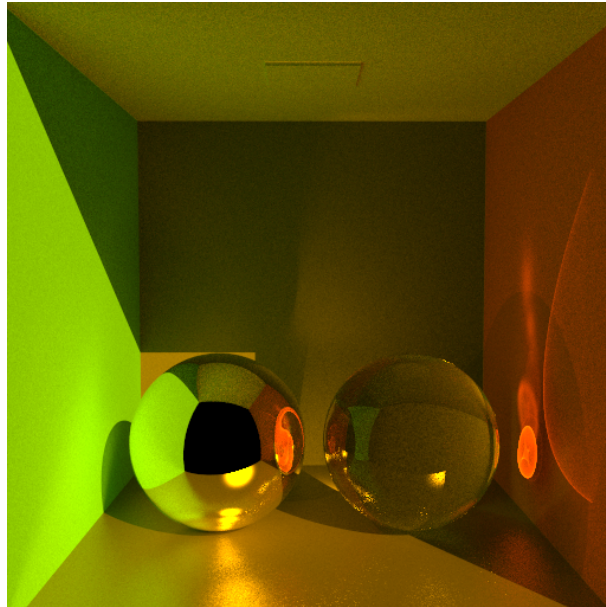


Figura 2.3: Uma imagem gerada pelo SmallVCM.

2.3 SmallVCM

SmallVCM (DAVIDOVIČ, 2012) é um renderizador de imagens, baseado em algoritmos de *ray tracing* e *path tracing*, desenvolvido pelo autor para servir como uma implementação de exemplo de seu algoritmo *Vertex Connection and Merging* (GEORGIEV et al., 2012). Além deste algoritmo, o programa também implementa uma variedade de outros incluindo *path tracing*, *light tracing*, *bidirectional path tracing* e *photon mapping*. Por ter propósito de educar sobre os algoritmos, não possui funcionalidades avançadas como animação e é limitado a renderizar um pequeno conjunto de cenas de exemplo incluídas no programa. Uma imagem gerada com o programa, utilizando o algoritmo de *vertex connection and merging*, pode ser vista na figura 2.3. O programa também possui uma função que gera um relatório de comparação da performance relativa de cada algoritmo.

Escrito em C++, o SmallVCM segue uma estrutura de projeto claramente organizada. A partir das opções passadas na linha de comando, é criada uma cena, utilizando um dos modelos disponíveis, e um renderizador, dos quais existem várias implementações para os vários algoritmos. Todos os renderizadores utilizam uma infra-estrutura de comum de tipos representando os objetos, fontes luminosas e materiais da cena. Os resultados intermediários da renderização são acumulados em um `Framebuffer` e subsequentemente salvos para um arquivo após um terem sido executadas um número especificado de iterações ou de certo tempo de execução.

Diagrama das classes/módulos.

2.3.1 Paralelização

Existe suporte à paralelização do processo de renderização. Este é implementado utilizando diretivas OpenMP (OpenMP Architecture Review Board, 2013), uma API multi-plataforma para computação paralela. A OpenMP permite a paralelização de código através da inserção de diretivas `#pragma omp` em programas C++, que são reconhecidas pelo compilador (portando requerendo suporte específico por parte deste) e automaticamente transformadas em código paralelo. Assim, uma aplicação existente pode ser mais facilmente convertida para um programa paralelo, sem depender de APIs de algum SO específico e sem precisar de grandes re-estruturações do programa nos casos mais simples.

No SmallVCM a paralelização é efetuada construindo-se várias instâncias do renderizador, uma por *thread*. Como cada iteração dos algoritmos de *path tracing* é independente da outra, estas podem ser executadas em paralelo. Assim, pode-se atingir um determinado número de iterações em menos tempo, sem necessitar modificar os algoritmos em si.

3 DESENVOLVIMENTO

Aqui são descritas as atividades realizadas para atingir os objetivos propostos. É dada uma explicação da organização do ambiente de compilação e desenvolvimento utilizadas nas versões do SmallVCM. (A original e a re-escrita em Rust, denominada SmallVCM-rs.) Seguindo, é feita uma descrição da organização de código do SmallVCM, e quais foram algumas diferenças marcantes encontradas entre as linguagens de como a implementação de seus componentes mudou no código Rust por influência de diferenças entre essa linguagem e C++.

3.1 Ambiente de Desenvolvimento e Organização de Projeto

3.1.1 Rust

O compilador `rustc`, além de ser oficialmente distribuído em forma de código fonte, também é disponibilizado através de pacotes binários distribuídos na página do projeto¹. Estes pacotes evitam que usuários passem pelo processo de *bootstrap* do compilador, que pode levar mais de uma hora, dependendo do hardware em que é executado. Atualizados diariamente, os pacotes binários são providos para os sistemas operacionais Linux, Mac OS X e Windows, em versões 32-bits ou 64-bits. Ocasionalmente são também feitas versões numeradas do compilador, porém o uso destas não é recomendado pela comunidade pois rapidamente ficam defasadas, um problema se pretende-se utilizar qualquer biblioteca de terceiros ou consultar a documentação online.

Para o desenvolvimento, foi utilizada uma máquina virtual contendo o sistema operacional Arch Linux. Além de conter software atualizado, existem pacotes de terceiros para a distribuição que automaticamente instalam versões atualizadas do compilador de Rust.² Como a linguagem ainda está em fase de mudança frequente, poder atualizar o compilador facilmente é importante para se manter atualizado com as últimas mudanças e funcionalidades incluídas diariamente na linguagem.

Além do compilador, também é utilizado o gerenciador de pacotes oficial Cargo³, desenvolvido especificamente para gerenciar e compilar bibliotecas Rust. Assim como o compilador,

¹ <http://www.rust-lang.org/install.html>

² <https://aur.archlinux.org/packages/rust-nightly-bin/>

³ <http://crates.io/>

também existem pacotes Arch Linux para manter este atualizado⁴.

Seguindo sua estrutura recomendada de projeto, é possível utilizar o Cargo para gerenciar a compilação, com o comando `cargo build`, evitando a necessidade de um *Makefile*. Para fazer uso desta funcionalidade, é necessário colocar todo o código fonte dentro de um subdiretório `src/`. O ponto de partida do compilador para fazer a descoberta de todo o código fonte deve ser em um arquivo chamado `main.rs` (para projetos executáveis) ou `lib.rs` (para bibliotecas). Fora deste diretório, deve ser criado um arquivo `Cargo.toml`, que contém metadados sobre o projeto e bibliotecas utilizadas. Esta estrutura de projeto padronizada facilita a familiarização de programadores com outros projetos Rust, já que não é necessário se adaptar a um sistema de compilação único a cada projeto.

Além das tarefas básicas de um sistema de compilação, o sistema também encarrega-se de automaticamente fazer o download e instalação de quaisquer bibliotecas externas utilizadas. Neste trabalho são utilizadas as bibliotecas *time* (Biblioteca oficial para utilizar as funções de relógio do sistema operacional.) e *rayon*. (Fornece uma primitiva simples para paralelização de tarefas.) Para inclusão de uma biblioteca externa, basta incluir a URL de um repositório Git no arquivo `Cargo.toml`, e ela será automaticamente baixada e compilada antes da compilação do programa.

Para realizar o controle de mudanças foi utilizado o Git⁵, também utilizado pelo projeto original.

3.1.2 C++

O SmallVCM original, sendo escrito em C++ precisa de um compilador para esta linguagem. Inicialmente o sistema de compilação vem configurado para utilizar-se o compilador gcc. O processo de compilação do programa é bastante simples, bastando executar `make` para compilar o binário. Não foram necessárias nenhuma modificação a versão distribuída do código.

3.2 Metodologia da *Port*

Estabeleceu-se um objetivo inicial de converter para Rust o conjunto de código necessário para renderizar uma imagem utilizando o algoritmo básico `EyeLight`, que calcula apenas a incidência de luz direta de um fonte luminosa posicionada no observador, em modo *single-*

⁴ <https://aur.archlinux.org/packages/cargo-nightly-bin/>

⁵ O repositório com o código pode ser encontrado em <https://github.com/yuriks/SmallVCM-rs>.

threaded. Isto possibilita que se tenha uma versão funcional para realizar testes de performance o mais cedo possível.

Procurou-se manter a organização do código da versão re-escrita em Rust similar à do original: Os arquivos mantêm o mesmo nome mas com a extensão `.cxx` ou `.hxx` substituída por `.rs`. Uma exceção é o arquivo `smallvcm.cxx`, que foi renomeado para `main.rs` para seguir a organização padronizada descrita na seção anterior.

Overview do código necessário para o objetivo.

3.3 Mudanças e Dificuldades

Durante o processo de re-escrita foram encontradas diversas dificuldades provenientes da escolha da Rust e de seu estado de desenvolvimento atual. Foram necessárias mudanças no paradigma de implementação de certos aspectos do programa para se adequar as convenções e funcionalidades da Rust.

3.3.1 Herança

Um dos elementos de linguagem notoriamente ausentes de Rust é o conceito de herança de tipos. As alternativas apresentadas para fazer re-uso de código e comportamentos polimórficos são o uso de composição e de *traits*, respectivamente.

No código do SmallVCM existem algumas instâncias de hierarquias de herança de classe, todas seguindo o mesmo padrão: Uma classe base abstrata, as vezes contendo alguns campos de dados concretos, além de definições de funções virtuais que devem ser substituídas pelas classes que as implementam. Esse padrão é utilizado pelas hierarquias definindo algoritmos de compilação (base `AbstractRenderer`); formas geométricas (base `AbstractGeometry`) e tipos de fontes luminosas (base `AbstractLight`).

Duas dessas hierarquias, de formas geométricas e de fontes luminosas, não possuem campos de dados na classe base e portanto podem ser diretamente convertidos para *traits* a serem implementados por estruturas.

Para realizar a conversão da hierarquia de `AbstractRenderer`, que possui campos de dados, para Rust, adotou-se a seguinte metodologia: Como a classe base contém campos de dados, é definida uma estrutura `RendererBase`, contendo estes campos compartilhados e quaisquer funções *não-virtuais* da classe base. É também definido um *trait* `AbstractRenderer` con-

tendo apenas as declarações das funções *virtuais* e também adicionando duas novas funções: `base` e `base_mut`. O propósito destas duas funções é retornar uma referência ou uma referência única, respectivamente, para uma instância `RendererBase` contida dentro do objeto, assim permitindo que código no resto do programa consiga pegar uma referência para as funções e dados da “classe base”. As classes que originalmente derivavam de `AbstractRenderer` no código C++ agora incluem um `RendererBase` como um de seus campos e implementam o *trait* para estender a estrutura/*trait* base. As duas implementações são demonstradas na figura 3.1.

O resultado desta conversão resulta num uso um tanto inconveniente do tipo, pois a base precisa ser explicitamente acessada quando se quer utilizar um campo, mas no geral não forçou nenhuma mudança arquitetural de grande porte.

3.3.2 Sobrecarga de Funções

Devido a falta de suporte a sobrecarga de funções, algumas funções tiveram que ser renomeadas durante o processo. No geral utilizou-se o mesmo nome da função original, com um sufixo indicando o tipo de parâmetro aceito pela função. Este ponto não gerou outras dificuldades.

3.3.3 Sobrecarga de Operadores e Conversões Automáticas

O módulo `math.rs` contém uma biblioteca de tipos matemáticos, consistindo de vetores 2D e 3D, além de matrizes 4x4. Este arquivo sofreu várias modificações comparado ao original devido ao seu frequente uso de sobrecarga de operadores, que diferem em sintaxe e implementação, e conversões automáticas, que não são suportadas em Rust.

Em Rust a sobrecarga de operadores é feita através da implementação de *traits* especiais. No entanto, este é um procedimento muito mais verboso do que simplesmente declarar funções com um nome especial como é feito em C++. Assim, para evitar uma grande quantidade de duplicação de código, foi usada a funcionalidade de macros de Rust para gerar as implementações dos operadores para `Vec2` e `Vec3` sem precisar repetir todas as definições. Parte da definição desta macro e sua correspondente expansão são mostradas na figura 3.2.

Outra dificuldade foi causada pela falta de conversões automáticas entre tipos. No código original ela era usada para converter escalares para vetores automaticamente, permitindo multiplicações vetor \times escalar e vetor \times vetor necessitando apenas de uma definição da operação de multiplicação. Para contornar isso, foram criadas funções `vec2s` e `vec3s` que realizam essa

```

class AbstractRenderer {
public:
    AbstractRenderer(const Scene& aScene) {
        // ...
    }

    virtual ~AbstractRenderer() {}
    virtual void RunIteration(int aIteration)
        = 0;

    void GetFramebuffer(Framebuffer&
        oFramebuffer) { /* ... */ }
    bool WasUsed() const { return mIterations
        > 0; }

public:
    uint mMaxPathLength;
    uint mMinPathLength;

protected:
    int mIterations;
    Framebuffer mFramebuffer;
    const Scene& mScene;
};

```

```

class EyeLight : public AbstractRenderer {
public:
    EyeLight(const Scene& aScene, int aSeed =
        1234)
        : AbstractRenderer(aScene), mRng(aSeed)
    {}

    virtual void RunIteration(int aIteration) {
        // ...
    }

    Rng mRng;
};

```

(a) C++

```

pub trait AbstractRenderer<'a> {
    fn base<'b>(&'b self) -> &'b
        RendererBase<'a>;
    fn base_mut<'b>(&'b mut self) -> &'b mut
        RendererBase<'a>;
    fn run_iteration(&mut self, iteration:
        u32);
}

pub struct RendererBase<'a> {
    pub max_path_length: u32,
    pub min_path_length: u32,

    // originally protected
    pub iterations: u32,
    pub framebuffer: Framebuffer,
    pub scene: &'a Scene,
}

impl<'a> RendererBase<'a> {
    pub fn new(scene: &Scene) -> RendererBase {
        // ...
    }

    pub fn get_framebuffer(&self) ->
        Framebuffer { /* ... */ }
    pub fn was_used(&self) -> bool {
        self.iterations > 0 }
}

```

```

pub struct EyeLight<'a> {
    base: RendererBase<'a>,
    rng: Rng
}

impl<'a> EyeLight<'a> {
    pub fn new(scene: &Scene, seed: u32)
        -> EyeLight {
        EyeLight {
            base: RendererBase::new(scene),
            rng: SeedableRng::from_seed([0, seed
                as u64]),
        }
    }
}

impl<'a> AbstractRenderer<'a> for
    EyeLight<'a> {
    fn base<'b>(&'b self) -> &'b
        RendererBase<'a> { &self.base }
    fn base_mut<'b>(&'b mut self) -> &'b mut
        RendererBase<'a> { &mut self.base }
    fn run_iteration(&mut self, iteration:
        u32) { /* ... */ }
}

```

(b) Rust

Figura 3.1: Classe abstrata AbstractRenderer e sua conversão para uma *struct* e *trait*.


```

// Definição das macros
macro_rules! impl_Vector_op(
    ($Trait:ident for $Self:ident { $($field:ident),+ }, $func:ident) => (
        impl<T: Num> $Trait<$Self<T>, $Self<T>> for $Self<T> {
            #[inline]
            fn $func(&self, o: &$Self<T>) -> $Self<T> {
                $Self {
                    $($field: self.$field.$func(&o.$field)),+
                }
            }
        }
    )
)

macro_rules! impl_Vector_traits(
    ($Self:ident { $($field:ident),+ }) => (
        impl_Vector_op!(Add for $Self { $($field),+ }, add)
        impl_Vector_op!(Sub for $Self { $($field),+ }, sub)
        impl_Vector_op!(Mul for $Self { $($field),+ }, mul)
        impl_Vector_op!(Div for $Self { $($field),+ }, div)

        impl<T: Num> Neg<$Self<T>> for $Self<T> {
            #[inline]
            fn neg(&self) -> $Self<T> {
                $Self {
                    $($field: -self.$field),+
                }
            }
        }
        // ...
    )
)

// Definição dos tipos
#[deriving(Copy, Clone)]
pub struct Vector2<T> { pub x: T, pub y: T }
#[deriving(Copy, Clone)]
struct Vector3<T> { pub x: T, pub y: T, pub z: T }

// Instanciação das implementações utilizando macros
impl_Vector_traits!(Vector2 { x, y })
impl_Vector_traits!(Vector3 { x, y, z })

```

Figura 3.2: Definição e uso de uma macro e sua expansão

conversão. No entanto, o código usuário precisa chama-las explicitamente.

Uma alternativa seria definir um *trait* `FromVector`, que poderia ser implementado por tipos escalares e vetores, convertendo os escalares para um vetor, e simplesmente retornaria o próprio vetor caso chamado nele. As implementações dos operadores seriam então genéricas sobre `T` que implementa `FromVector` e chamariam a função de conversão. Esta técnica é utilizada em alguns lugares na biblioteca padrão quando é necessário realizar conversão de uma variedade de tipos diferentes de forma automática.

3.3.4 Instabilidade

Rust ainda está em um período de intenso desenvolvimento e por isso mudanças na definição da linguagem e na implementação do compilador ainda acontecem frequentemente. Foram encontradas várias situações onde uma atualização do compilador, buscando arrumar *bugs* de compilação ou tomar proveito de funções novas da linguagem, necessitou a atualização do código para se adequar a revisões feitas na linguagem ou na biblioteca padrão.

3.3.5 *Enums*

O módulo de configuração (`config.rs`) faz uso de uma *enum* para selecionar o algoritmo. Em C++, valores de uma enumeração tem um respectivo valor inteiro associado. Rust não associa valores inteiros correspondentes a cada valor da enumeração, pois valores podem ter outros itens de dados associados a eles, sendo análogos a Tipos de Dados Algébricos presentes em Haskell ou Standard ML. Portanto, algumas técnicas utilizadas pelo código original, como iterar através dos possíveis valores numéricos das enumerações ou as utilizar como índice em um vetor, não são possíveis aqui. Estes usos são substituídos por usos da expressão `match`, que por requerer a menção dos nomes dos itens da enumeração é mais repetitiva, mas também mais clara e robusta, pois os itens da enumeração não correm risco de sair fora de sincronia com os índices do vetor. Uma comparação entre os dois tipos de código pode ser vista na figura 3.3. Nota-se que os testes para verificar se o valor está dentro do intervalo esperado não são necessários em Rust, pois a linguagem não permite a atribuição de valores não presentes na enumeração a variáveis desse tipo.

Esta funcionalidade também foi utilizada de maneira vantajosa, substituindo um par de variáveis de configuração que determinava a condição de parada do algoritmo, dos quais apenas uma poderia estar ativa de cada vez, por uma *enum* de duas variantes, cada uma contendo seu

```

enum Algorithm {
    kEyeLight,
    kPathTracing,
    kLightTracing,
    kProgressivePhotonMapping,
    kBidirectionalPhotonMapping,
    kBidirectionalPathTracing,
    kVertexConnectionMerging,
    kAlgorithmMax
};

static const char* GetName(Algorithm
    aAlgorithm) {
    static const char* algorithmNames[7] = {
        "eye light",
        "path tracing",
        "light tracing",
        "progressive photon mapping",
        "bidirectional photon mapping",
        "bidirectional path tracing",
        "vertex connection and merging"
    };

    if(aAlgorithm < 0 || aAlgorithm > 7)
        return "unknown algorithm";

    return algorithmNames[aAlgorithm];
}

static const char* GetAcronym(Algorithm
    aAlgorithm) {
    static const char* algorithmNames[7] = {
        "el", "pt", "lt", "ppm", "bpm", "bpt",
        "vcm" };

    if(aAlgorithm < 0 || aAlgorithm > 7)
        return "unknown";
    return algorithmNames[aAlgorithm];
}

```

(a) C++

```

enum Algorithm {
    EyeLight,
    PathTracing,
    LightTracing,
    ProgressivePhotonMapping,
    BidirectionalPhotonMapping,
    BidirectionalPathTracing,
    VertexConnectionMerging,
}

impl Algorithm {
    fn get_name(self) -> &'static str {
        match self {
            EyeLight => "eye light",
            PathTracing => "path tracing",
            LightTracing => "light tracing",
            ProgressivePhotonMapping =>
                "progressive photon mapping",
            BidirectionalPhotonMapping =>
                "bidirectional photon mapping",
            BidirectionalPathTracing =>
                "bidirectional path tracing",
            VertexConnectionMerging => "vertex
                connection and merging",
        }
    }

    fn get_acronym(self) -> &'static str {
        match self {
            EyeLight => "el",
            PathTracing => "pt",
            LightTracing => "lt",
            ProgressivePhotonMapping => "ppm",
            BidirectionalPhotonMapping => "bpm",
            BidirectionalPathTracing => "bpt",
            VertexConnectionMerging => "vcm",
        }
    }
}

```

(b) Rust

Figura 3.3: Comparação entre *enums* em C++ e Rust

respectivo parâmetro. Esta mudança evita que o sistema acidentalmente seja colocado em um estado inválido, e também remove ambiguidade na hora de testar pela condição.

3.3.6 Ponteiros Nulos

Como Rust desestimula o uso de ponteiros, programadores devem geralmente substituí-los por referências, que não podem ser nulas e que também estão são verificadas em compilação para garantir que nunca se tornem inválidas, ou utilizar o tipo `Box<T>`, que representa uma alocação de memória pertencente a uma variável ou campo e que é gerenciada automaticamente para manter estas mesmas garantias.

O código original inicializava alguns ponteiros para nulo. Em sua maioria estes casos foram simplesmente adaptados para diretamente inicializar o ponteiro com seu valor final,

agora com a garantia de que sempre terão um valor válido. Em um caso foi utilizado o tipo `Option<T>`⁶ para permitir uma alocação opcional de um valor. Este uso também poderia ser eliminado com alguma re-estruturação do código.

3.3.7 Biblioteca padrão

É necessário de um gerador de números pseudo-aleatórios para o cálculo da integração por monte carlo. O SmallVCM original utilizava o popular *MT19937* (MATSUMOTO; NISHIMURA, 1998), presente na biblioteca padrão C++. Como este gerador não estava presente na biblioteca padrão da Rust, foi optado implementar o algoritmo *xorshift128+* (VIGNA, 2014) devido a seu bom desempenho, boa qualidade de resultados e pequena implementação quando comparado ao *MT19937*. O algoritmo também foi posteriormente implementado em C++ para que ambos possuíssem as mesmas características para fins de comparação.

Para a leitura de parâmetros passado via linha de comando, foi utilizado o módulo `getopts`, presente na biblioteca padrão. Na versão C++ a leitura é feita de forma manual.

⁶ Um *enum* fornecido na biblioteca padrão que possui dois estados: preenchido com um valor `T` ou vazio.

4 RESULTADOS

Ao fim do trabalho, havia sido portado para Rust código suficiente do SmallVCM para executar o algoritmo `EyeLight`. Funções que não eram necessárias para este caminho de execução não foram definidas ou foram definidas como *placeholders* que abortam o programa quando executadas, assim certificando que não são chamadas durante o teste. Não foram feitas mudanças algorítmicas, mas detalhes da implementação foram ajustados a medida que isto era requerido pela linguagem, como descrito no capítulo anterior, ou para produzir código mais idiomático se isto pudesse ser feito sem re-estruturações significativas da estrutura do programa. Quando executados com os mesmos parâmetros, as saída da *port* é fiel a do original.

4.1 Testes de Desempenho

A fim de averiguar se pode se esperar uma diferença de performance quando se implementa um algoritmo em Rust comparado a um equivalente em C++, foram feitas medidas do tempo de execução da mesma tarefa nas duas implementações.

Foi feita uma tentativa de se utilizar o compilador clang, que, assim como o rustc, utiliza o LLVM como *back-end* de otimização e geração de código, a fim de eliminar esta diferença e tornar as comparações com a versão escrita em Rust mais diretamente aplicáveis. Para isso, foi necessário remover o suporte a OpenMP do programa, pois a tecnologia não é bem suportada pelo clang. Esta modificação necessitou de poucas mudanças, bastando a remoção de algumas diretivas `#include` e `#pragma omp` e chamadas de inicialização do OpenMP.⁷ Após estas modificações o programa pôde ser compilado e executado com sucesso, mas sua execução mostrou-se demorada, com o clang gerando código muito menos eficiente que o gcc e rustc, resultando em uma diferença de aproximadamente 300% no tempo de execução. Como Rust, assim como clang, também utiliza o LLVM para sua geração de código, eram esperados resultados similares. Investigações realizadas com desenvolvedores do LLVM em seu canal IRC indicaram que seria vantajoso tentar a última versão de desenvolvimento (*Head of Tree*) do clang, que havia recebido melhoras na geração de código em áreas relevantes ao programa. Fazendo uso da versão de desenvolvimento, constatou-se que ambos compiladores agora geravam código com desempenho similar.

⁷ Esta versão, também incluindo a implementação do PRNG *xorshift128+*, pode ser encontrada em <https://github.com/yuriks/SmallVCM/tree/xorshift>.

Iterações	Threads	rustc	clang	gcc
100	1	7.665 s (85.45%)	8.253 s (92%)	8.970 s (100%)
400	4	8.802 s (84.59%)	—	10.405 s (100%)

Tabela 4.1: Tempo de execução do algoritmo `EyeLight` em diferentes implementações e compiladores. As porcentagens indicam o tempo relativo em relação a versão mais lenta.

Todas as medições foram realizadas num computador com a seguinte configuração de hardware: CPU Intel i5-4670K 3.4 GHz Quad-core, 16 GB de RAM e disco Samsung 840 Pro 256 GB. O SO utilizado foi Arch Linux, arquitetura `x86_64`, rodando dentro de uma máquina virtual VirtualBox com 3GB de RAM, 4 cores e 40 GB de disco, com Windows 8.1 como SO *host*. As versões dos compiladores utilizadas foram: `gcc 4.9.2`, `clang SVN 222777`, `rustc Git 81eeec094`. As opções de compilação utilizadas foram: `-O3 -std=c++11` (`gcc`), `-O3 -std=c++11 -stdlib=libc++ -lc++abi` (`clang`) e `--opt-level 3` (`rustc`). Para a versão *multi-threaded* no compilador `gcc`, foi também adicionada a opção `-fopenmp` para habilitar o suporte a OpenMP.

Para medir o tempo de execução do algoritmo, foi utilizada a ferramenta `perf stat`. Passando-se a opção `-r` a ferramenta irá repetir o teste várias vezes, reportando a média dos dados coletados. Esta função foi utilizada com 10 repetições em cada teste. A invocação do `SmallVCM` foi feita com as opções `-a el -i 100` no teste *single-threaded* e `-a el -i 400` no teste *multi-threaded*. Os resultados obtidos estão dispostos na tabela 4.1.

Os tempos e as diferenças entre eles são consistentes através de várias execuções dos testes e mantém-se proporcionais ao número de iterações realizadas quando este parâmetro é alterado.

5 CONCLUSÃO

TODO

REFERÊNCIAS

- COOK, R. L.; PORTER, T.; CARPENTER, L. Distributed Ray Tracing. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.18, n.3, p.137–145, Jan. 1984.
- DAVIDOVIČ, T. **SmallVCM**: a (not too) small physically based renderer. <http://www.smallvcm.com/>, acessado em Agosto de 2014.
- GEORGIEV, I. et al. Light Transport Simulation with Vertex Connection and Merging. **ACM Trans. Graph.**, New York, NY, USA, v.31, n.6, p.192:1–192:10, Nov. 2012.
- GROSSMAN, D. et al. Region-based Memory Management in Cyclone. **SIGPLAN Not.**, New York, NY, USA, v.37, n.5, p.282–293, May 2002.
- HOLK, E. et al. GPU Programming in Rust: implementing high-level abstractions in a systems-level language. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS PHD FORUM (IPDPSW), 2013 IEEE 27TH INTERNATIONAL. **Anais...** [S.l.: s.n.], 2013. p.315–324.
- KAJIYA, J. T. The Rendering Equation. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.20, n.4, p.143–150, Aug. 1986.
- LAFORTUNE, E. P.; WILLEMS, Y. D. Bi-directional path tracing. In: COMPUGRAPHICS. **Proceedings...** [S.l.: s.n.], 1993. v.93, p.145–153.
- LATTNER, C.; ADVE, V. LLVM: a compilation framework for lifelong program analysis transformation. In: CODE GENERATION AND OPTIMIZATION, 2004. CGO 2004. INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2004. p.75–86.
- MATSUMOTO, M.; NISHIMURA, T. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. **ACM Trans. Model. Comput. Simul.**, New York, NY, USA, v.8, n.1, p.3–30, Jan. 1998.
- MILNER, R. **The Definition of Standard ML**: revised. [S.l.]: MIT Press, 1997.
- Mozilla Research. **The Rust Programming Language**. <http://rust-lang.org/>, acessado em Agosto de 2014.

Mozilla Research. **The Servo Browser Engine**. <https://github.com/servo/servo>, acessado em Agosto de 2014.

OpenMP Architecture Review Board. **OpenMP Application Program Interface, Version 4.0**. <http://openmp.org/wp/openmp-specifications/>, acessado em Novembro de 2014.

PHARR, M.; HUMPHREYS, G. **Physically Based Rendering, Second Edition**: from theory to implementation. 2nd.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

POUZANOV, V. **Zinc, the bare metal stack for Rust**. 2014.

Rust Project Developers. **The Rust References and Lifetimes Guide**. <http://doc.rust-lang.org/guide-lifetimes.html>, acessado em Novembro de 2014.

Rust Project Developers. **The Rust Guide**. <http://doc.rust-lang.org/guide.html>, acessado em Novembro de 2014.

Rust Project Developers. **Writing Unsafe and Low-Level Code in Rust**. <http://doc.rust-lang.org/guide-unsafe.html>, acessado em Novembro de 2014.

SCHÄRLI, N. et al. Traits: composable units of behaviour. In: CARDELLI, L. (Ed.). **ECOOP 2003 – Object-Oriented Programming**. [S.l.]: Springer Berlin Heidelberg, 2003. p.248–274. (Lecture Notes in Computer Science, v.2743).

VEACH, E. **Robust Monte Carlo methods for light transport simulation**. 1997. Tese (Doutorado em Ciência da Computação) — Stanford University.

VIGNA, S. Further scramblings of Marsaglia’s xorshift generators. **CoRR**, [S.l.], v.abs/1404.0390, 2014.

WADLER, P.; BLOTT, S. How to Make Ad-hoc Polymorphism Less Ad Hoc. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 16., New York, NY, USA. **Proceedings...** ACM, 1989. p.60–76. (POPL ’89).

WHITTED, T. An Improved Illumination Model for Shaded Display. **Commun. ACM**, New York, NY, USA, v.23, n.6, p.343–349, June 1980.