Exploração da Linguagem Rust para o Desenvolvimento de um *Path Tracer* Paralelo

Yuri Kunde Schlesner
Universidade Federal de Santa Maria

15 de agosto de 2014

1 Identificação

Resumo:

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 quer 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.

Período de execução: Setembro de 2014 a Dezembro de 2014

Unidades participantes:

Curso de Ciência da Computação
Departamento de Eletrônica e Computação

Área de conhecimento: Ciência da Computação

Linha de Pesquisa: Computação Gráfica, Linguagens de Programação, Programação Paralela

Tipo de projeto: Trabalho de Conclusão de Curso

Participantes:

Profª Andrea Schwertner Charão – Orientadora
Yuri Kunde Schlesner – Orientando

2 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: É 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 a limitada capacidade computacional disponível, eram utilizadas aproximações grosseiras que, embora produzissem imagens atrativas, não eram muito realísticas, 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 a sua habilidade de simular o comportamento da luz com relativa precisão.

No entanto, este realismo vem ao custo de muito poder de processamento, e 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 as penalidades de performance 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 mais fácil desenvolvimento mas sem sacrificar a performance requerida.

A linguagem de programação *Rust*, 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 alta-performance, segurança e expressividade. Ela atinge isso usando um modelo tradicional de compilação prévia (*ahead of time*) e um sistema de tipos que permite a verificação automática dos usos de ponteiros durante a compilação, eliminando a possibilidade de acontece-

rem erros de memória sem introduzir penalidades excessivas de performance 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.

3 Objetivos

3.1 Objetivo Geral

O objetivo geral deste trabalho é portar um *path tracer* para Rust e, através deste processo, realizar uma comparação qualitativa e quantitativa entre esta linguagem e C++, nos aspectos de performance e organização de código. Como base será utilizado o SmallVCM[2], um *path tracer* de propósito educativo escrito em C++, escolhido por implementar uma variedade de algoritmos diferentes de *path tracing* e por ser relativamente compacto, consistindo de aproximadamente 5000 linhas de código.

3.2 Objetivos Específicos

- Estudar o código original do SmallVCM.
- Re-escrever um subconjunto mínimo do SmallVCM utilizando Rust, para a realização de testes.
- Realizar uma comparação de performance e clareza de código entre as duas versões e identificar possíveis melhorias.
- Paralelizar o renderizador, afim de melhorar sua performance, fazendo uso das funcionalidades de Rust.
- Portar o restante do SmallVCM, afim de que as duas versões tenham funcionalidades equivalentes.

4 Justificativa

Rust é uma linguagem relativamente nova e, embora aplicações gráficas de alta performance 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.

5 Revisão de Literatura

5.1 *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 a 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.)[10]

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 [15] 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 et al. [1] 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, 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 [4] 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 1.) 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

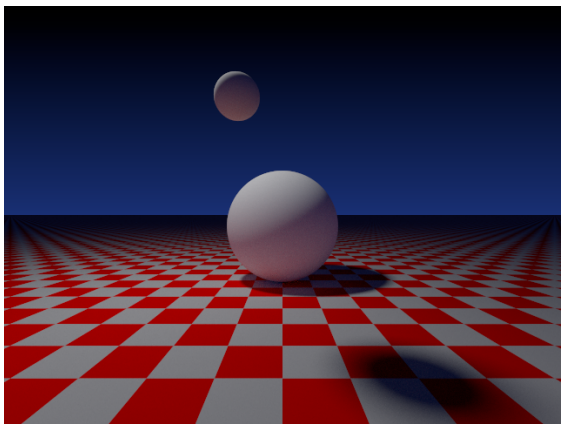

Figura 1: Um exemplo de uma imagem gerada utilizando *path tracing*. Note como a luz que atinge o plano xadrez é refletida de volta para iluminar a esfera, um fenômeno conhecido como *iluminação indireta* e que é corretamente simulado pelo algoritmo.

por Cook et al. 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 and Willems [5]. Este algoritmo foi depois reformulado por Veach em [14], onde também introduziu a técnica de *multiple importance sampling* e o algoritmo *Metropolis light transport*.

5.2 Rust

Rust[8] é uma linguagem de programação de propósito geral, oficialmente patrocinada pela *Mozilla Research* mas desenvolvida também pela comunidade. É uma linguagem de propósito geral, que busca atender as necessidades de programadores que atualmente utilizariam linguagens como C ou C++, por questões de performance ou de controle sobre o hardware. Uma de suas características mais distintivas é o modelo de referências que utiliza, baseado em regiões[13], 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 foi originalmente desenvolvida para servir como linguagem de implementação do projeto Servo[9], um browser experimental de próxima geração também desenvolvido pela Mozilla Research, 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, trazendo uma forte inspiração de linguagens como ML[7]. Assim, possui funcionalidades como *pattern matching*, tuplas, *sum types* (também conhecidos como *Algebraic Data Types* (ADTs) ou *uniões discriminadas*.) A linguagem não implementa o modelo tradicional de programação orientada a objetos, oferecendo em troca o conceito de *traits* (similares ao conceito de *type classes* em Haskell ou de *interfaces* em outras linguagens) que evita alguns problemas do modelo baseado em herança de classes, como a falta de extensibilidade de tipos e o *problema diamante* em herança múltipla[12].

O compilador oficial da linguagem, o *rustc*, é implementado na própria linguagem e utiliza a infraestrutura do projeto LLVM[6], 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[3], que permite a compilação de código Rust para execução paralela em GPUs, e Zinc[11], um *framework* para desenvolvimento de aplicações embarcadas em microprocessadores.

6 Metodologia

Dado seu caráter prático, de implementar um programa em uma nova linguagem, esta pesquisa se enquadra como pesquisa aplicada. Como o objetivo será de explorar melhores práticas e as possíveis vantagens conferidas pela linguagem durante este processo, é uma pesquisa exploratória.

7 Plano de Atividades e Cronograma

1. **Estudar o código original do SmallVCM:** Será feito um estudo do SmallVCM original, afim de aprender como é organizado e como pode ser melhor expresso em Rust.

2. **Re-escrever um subconjunto mínimo do SmallVCM utilizando Rust**: Para economizar tempo nesta etapa, serão portados apenas os algoritmos essenciais necessários para a renderização de imagens. O restante do programa será portado depois durante a etapa 5.

3. **Realizar uma comparação entre as duas versões**: As duas versões do renderizador, a nova feita em Rust e a antiga em C++, serão comparadas nos quesitos de performance e de organização do código, afim de identificar avanços ou regressos, e se esses podem ser atribuídos a características de cada linguagem.

4. **Paralelizar o renderizador**: A nova versão do renderizador será paralelizada, afim de fazer melhor uso do hardware. Para atingir este objetivo serão exploradas todas as alternativas oferecidas pela linguagem, não se limitando a um tipo específico de paralelização.

5. **Portar o restante do SmallVCM**: O trabalho iniciado na etapa 2 será finalizado com a escrita do restante do SmallVCM, afim de que a versão em Rust implemente a mesmas funcionalidades que a original. De acordo com a disponibilidade de tempo, este código novo também sera paralelizado.

Espera-se que o desenvolvimento das atividades siga o seguinte cronograma:

Etapa	Setembro	Outubro	Novembro	Dezembro
1	✓			
2	✓	✓		
3		✓		✓
4		✓	✓	✓
5			✓	✓

Tabela 1: Cronograma de Atividades

8 Recursos

Para a realização deste trabalho será utilizado apenas equipamento pessoal do pesquisador, visto que não é necessário o uso de qualquer equipamento especial além de um computador para desenvolvimento.

9 Resultados Esperados

Ao término deste trabalho, espera-se ter uma implementação de um renderizador, escrito em Rust, que seja capaz de produzir imagens fotorrealistas, fazendo uso eficiente do paralelismo oferecido pelo hardware.

Referências

[1] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, Jan. 1984. ISSN 0097-8930. doi: 10.1145/964965.808590.

[2] T. Davidovič. SmallVCM: A (not too) small physically based renderer, 2012. URL `http://www.smallvcm.com`.

[3] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis. 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*, pages 315–324, May 2013. doi: 10.1109/IPDPSW.2013.173.

[4] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, Aug. 1986. ISSN 0097-8930. doi: 10.1145/15886.15902.

[5] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. In *Proceedings of CompuGraphics*, volume 93, pages 145–153, 1993.

[6] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665.

[7] R. Milner. *The Definition of Standard ML: Revised*. MIT Press, 1997. ISBN 9 780 262 631 815.

[8] Mozilla Research. The Rust programming language, . URL `http://rust-lang.org/`.

[9] Mozilla Research. The Servo browser engine, . URL `https://github.com/servo/servo`.

[10] M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. ISBN 0 123 750 792, 9 780 123 750 792.

[11] V. Pouzanov. Zinc, the bare metal stack for rust. URL `http://zinc.rs/`.

[12] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40 531-3. doi: 10.1007/978-3-540-45070-2_12.

[13] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997. ISSN 0890-5401. doi: 10.1006/inco.1996.2613.

[14] E. Veach. *Robust Monte Carlo methods for light transport simulation.* PhD thesis, Stanford University, 1997.

[15] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980. ISSN 0001-0782. doi: 10.1145/358876.358882.