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: Profª. Drª. 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 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.

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

SUMÁRIO

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: 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 acontecerem 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.

1.1 Objetivos

1.1.1 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(DAVIDOVIč, 2012), 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.

1.1.2 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.

1.2 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.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

Neste capítulo será 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. 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(TOFTE; TALPIN, 1997), 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(Mozilla Research, 2012), 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(MILNER, 1997). 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(SCHäRLI et al., 2003).

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.

2.2 *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 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.)(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, 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 2.1.) 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 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 por VEACH em (VEACH, 1997), onde também introduziu a técnica de *multiple importance sampling* e o algoritmo *Metropolis light transport*.

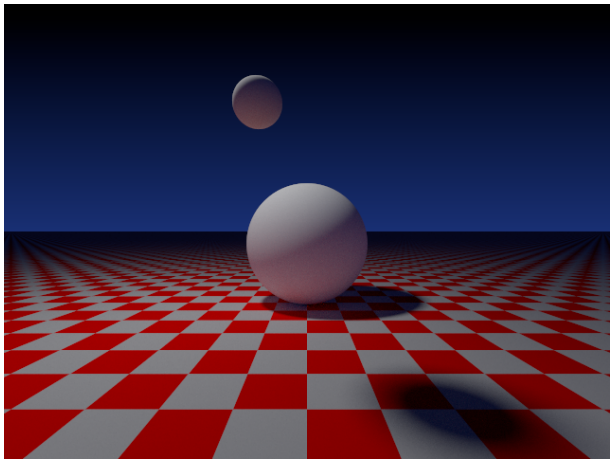

Figura 2.1 – Um exemplo de uma imagem gerada utilizando *path tracing*. Note como a luz que atinge o plano xadrez é refletida de volta para iluminar a esfera, um fenômeno conhecido como *iluminação indireta* e que é corretamente simulado pelo algoritmo.

3 DESENVOLVIMENTO

Aqui serão descritas as atividades realizadas para atingir os objetivos propostos. Será 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, será feita uma descrição da organização de código do *SmallVCM*, e 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 projeto[1]. Estes pacotes evitam que usuários passem pelo processo de *bootstrap* do compilador, que pode levar de uma à varias horas, 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 seu uso não é recomendado pois rapidamente ficam defasadas, um problema se pretende-se utilizar qualquer biblioteca de terceiros.

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 *Cargo*[3], desenvolvido especificamente para gerenciar e compilar bibliotecas *Rust*. Assim como o compilador, também existem pacotes *Arch Linux* para manter este atualizado[4].

[1] http://www.rust-lang.org/install.html
[2] https://aur.archlinux.org/packages/rust-nightly-bin/
[3] http://crates.io/
[4] https://aur.archlinux.org/packages/cargo-nightly-bin/

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*. Além das tarefas básicas de um build system, o sistema também encarrega-se de automaticamente fazer o download e instalação de quaisquer bibliotecas externas utilizadas. Neste trabalho, no entanto, esta funcionalidade não é utilizado, pois não será feito o uso de nenhuma biblioteca.

Para fazer uso da funcionalidade de compilação do *Cargo*, 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.

Para realizar o controle de mudanças foi utilizado o *Git*[5], também utilizado pelo projeto original.

3.1.2 C++

O *SmallVCM* original, sendo escrito em C++ precisa de um compilador para esta linguagem. No momento está sendo utilizado o *gcc*. O processo de compilação do programa é bastante simples, bastando executar `make` para compilar o binário. Não foram necessárias nenhumas modificações a versão distribuída do código.

No futuro planeja-se trocar para o compilador *clang* que, assim como o *rustc*, utiliza o *LLVM* como *back-end* de otimização e geração de código, afim de tornar as comparações com a versão escrita em *Rust* mais diretas. Isto irá requerer algumas modificações ao fonte para permitir a compatibilidade com o outro compilador.

3.2 Organização do Código

Procurou-se manter a organização do código da versão re-escrita em *Rust* similar à do original: Os arquivos mantém o mesmo nome, com a extensão `.cxx` ou `.hxx` substituída por `.rs`. Uma exceção é o arquivo `smallvcm.cxx`, que foi renomeado para `main.rs` de acordo

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

com a organização padronizada descrita na seção anterior.

Nesta seção será dado uma visão geral de como o código do *SmallVCM-rs* é organizado, divido por arquivo fonte. Também serão ressaltadas mudanças relevantes que foram realizadas nele comparando-se com o *SmallVCM*.

3.2.1 `config.rs`

Este arquivo lida com o gerenciamento da configuração do renderizador. Como o renderizador suporta apenas algumas cenas fixas, isto envolve principalmente a leitura de opções de configuração passadas na linha de comando. Ela é responsável por criar as instancias de `Scene`, `AbstractRenderer` e `Framebuffer` e retorna-las para serem usadas pelo resto do código.

Neste módulo, uma diferença que afeta o código significantemente é a diferença nas implementações de *enums* (tipos enumeráveis). Ao contrário de em C++, onde são pouco mais que um conjunto de constantes simbólicas mapeadas a inteiros, *enums* em *Rust* não tem um valor inteiro correspondente a cada valor da enumeração. De fato, valores da enumeração podem ter outros itens de dados distintos associados, sendo mais similares a Tipos de Dados Algébricos presentes em *Haskell* ou *Standard ML*. Portanto, algumas técnicas utilizadas pelo código original, que iterava através dos possíveis valores numéricos das enumerações ou as utilizava como índice em um vetor, não são possíveis aqui. Estes usos são substituídos por usos da expressão `match`, mais repetitiva mas também mais clara. Uma comparação entre os dois tipos de código pode ser vista na figura 3.1. Note como 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.

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

3.2.2 `framebuffer.rs`

Este arquivo contem a estrutura `Framebuffer`, que tem como objetivo armazenar as amostras geradas pela renderização, e depois combiná-las para exportar uma imagem.

Uma mudança realizada aqui é a eliminação do construtor padrão da classe. *Rust* não permite que membros de estruturas tenham valores não inicializados, portanto foi mantido apenas a função `setup`, que originalmente fazia a inicialização dos valores, e agora também cria

```
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];
}
```

```
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",
    }
  }
}
```

Figura 3.1 – Comparação entre *enums* em C++ e *Rust*

a instância da estrutura.

3.2.3 `main.rs`

Este é o ponto de entrada principal do programa. Como este é o ponto de partida da compilação, todos os outros arquivo fonte precisam ser referenciados aqui utilizando declarações `mod`. O compilador então os lê e os inclui no processo de compilação.

3.2.4 `math.rs`

Aqui são definidas classes matemáticas, consistindo de vetores 2D e 3D, além de matrizes 4x4. Este arquivo sofreu várias modificações devido a funcionalidades de C++ que não estão presentes ou são diferentes em *Rust*, particularmente sobrecarga de operadores, sobrecarga de funções e conversões automáticas.

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, é 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.

Outras dificuldades são causadas pela falta de sobrecarga de funções[6], que requer que funções com nomes iguais sejam renomeadas para desambiguar-las. A falta de conversões automáticas é especialmente onerosa: No código original ela era usada para converter escalares para vetores automaticamente, permitindo a multiplicação vetor $\times$ escalar, além de vetor $\times$ vetor. Para contornar isso, foram criadas funções `vec2s` e `vec3s` que realizam essa conversão. No entanto, o código usuário precisa chama-las explicitamente.

3.2.5 `renderer.rs`

Este arquivo contém a classe base `AbstractRenderer`, que é a base que os vários algoritmos de renderização presentes no programa implementam. No caso do *Rust*, esta classe é representada por um *trait*. Como a classe original continha alguns campos de dados e *Rust* não

[6] Suporte a *despache múltiplo* em *traits* é uma funcionalidade planejada que pode permitir algo equivalente no caso de sobrecarga de operadores. Ver https://github.com/rust-lang/rfcs/pull/195

```
// Definição da macro
macro_rules! impl_Vector_traits(
  ($Self:ident { $($field:ident),+ }) => (
    impl<T: Num> Add<$Self<T>, $Self<T>> for $Self<T> {
      #[inline]
      fn add(&self, o: &$Self<T>) -> $Self<T> {
        $Self {
          $($field: self.$field + o.$field),+
        }
      }
    }
    // ...
    impl<T: Num> Neg<$Self<T>> for $Self<T> {
      #[inline]
      fn neg(&self) -> $Self<T> {
        $Self {
          $($field: -self.$field),+
        }
      }
    }
    // ...
    impl<T: Num> Index<uint, T> for $Self<T> {
      #[inline]
      fn index(&self, index: &uint) -> &T {
        [$(&self.$field),+][*index]
      }
    }
  )
)

// 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 da macro
impl_Vector_traits!(Vector2 { x, y })
impl_Vector_traits!(Vector3 { x, y, z })
```

```
// Expansão de impl_Vector_traits!(Vector2 { x, y })
impl<T: Num> Add<Vector2<T>, Vector2<T>> for Vector2<T> {
  #[inline]
  fn add(&self, o: &Vector2<T>) -> Vector2<T> {
    Vector2 {
      x: self.x + o.x,
      y: self.y + o.y
    }
  }
}
// ...
impl<T: Num> Neg<Vector2<T>> for Vector2<T> {
  #[inline]
  fn neg(&self) -> Vector2<T> {
    Vector2 {
      x: -self.x,
      y: -self.y
    }
  }
}
// ...
impl<T: Num> Index<uint, T> for Vector2<T> {
  #[inline]
  fn index(&self, index: &uint) -> &T {
    [self.x, self.y][*index]
  }
}
```

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

implementa herança de estruturas[7], estes são separados em uma estrutura `RendererBase`. Tipos que implementam `AbstractRenderer` devem conter umas instância desta estrutura e disponibiliza-la através de uma chamada no *trait*.

[7] Também espera-se que alguma forma de herança eventualmente seja incluída na linguagem, mas não há ainda propostas aceitas.

4 PRÓXIMAS ETAPAS

Como continuação do trabalho realizado até o momento, são planejadas as seguintes atividades:

1. Continuar a re-escrita em andamento do SmallVCM, afim de que esta tenha funcionalidade o suficiente para gerar imagens usando pelo menos um tipo de algoritmo.

2. Paralelizar o código do renderizador, explorando diferentes alternativas para atingir este fim.

3. Realizar testes comparativos de performance para determinar a performance relativa entre duas versões, como parte do objetivo de determinar se a linguagem *Rust* é compatível com este tipo de aplicação.

4. Trazer o resto dos algoritmos de renderização do SmallVCM à versão em *Rust*, para que ambos tenham o mesmo conjunto de funções.

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.

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.

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.

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.

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).

TOFTE, M.; TALPIN, J.-P. Region-Based Memory Management. **Information and Computation**, [S.l.], v.132, n.2, p.109 – 176, 1997.

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

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