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

Yuri Kunde Schlesner
Orientador: Profª Drª Andrea Scwertner Charão

Ciência da Computação
Universidade Federal de Santa Maria

20/10/2014

informáticaufsm
Ciência da Computação

Outline

1. Introdução
   - Objetivos
   - Justificativa
2. Fundamentação
   - Rust
3. Desenvolvimento
   - Implementação
4. Próximos Passos
   - Terminar a port
   - Paralelização

informáticaufsm
Ciência da Computação

Introdução

- Ferramenta utilizada: Rust
  - Princípios e público-alvo
- Área de aplicação: Path Tracing
  - Requisitos de processamento

informáticaufsm
Ciência da Computação

Outline

1. Introdução
   - Objetivos
   - Justificativa
2. Fundamentação
   - Rust
3. Desenvolvimento
   - Implementação
4. Próximos Passos
   - Terminar a port
   - Paralelização

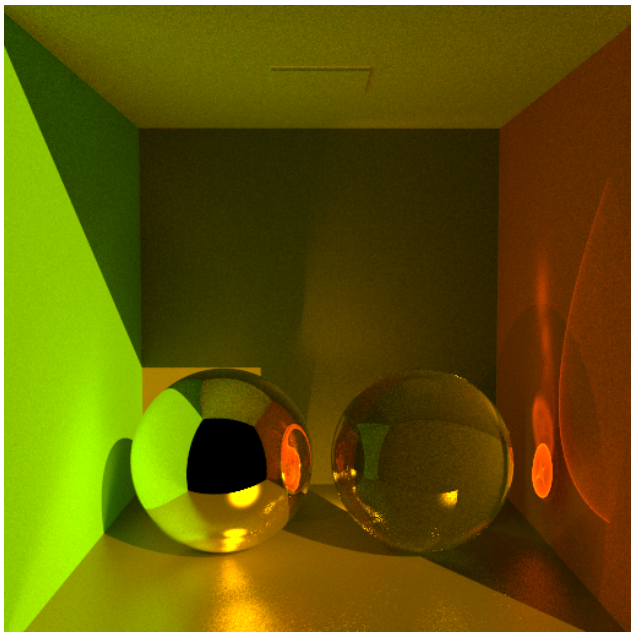
informáticaufsm
Ciência da Computação

Objetivos

- Portar um renderizador C++ para Rust

Objetivos

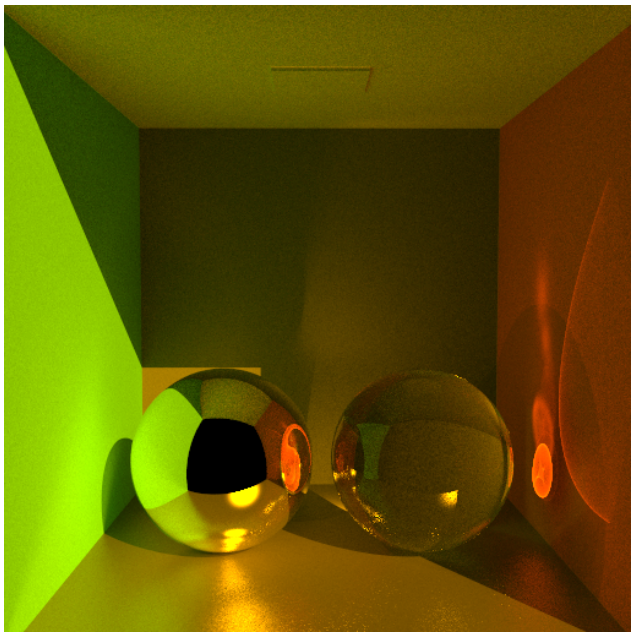
- Portar um renderizador C++ para Rust
  - SmallVCM
  - Pequeno: $\sim$5000 linhas de código



informáticaufsm
Ciência da Computação

Objetivos

- Portar um renderizador C++ para Rust
  - SmallVCM
  - Pequeno: $\sim$5000 linhas de código
- Paralelização



informáticaufsm
Ciência da Computação

Outline

1. Introdução
   - Objetivos
   - Justificativa
2. Fundamentação
   - Rust
3. Desenvolvimento
   - Implementação
4. Próximos Passos
   - Terminar a port
   - Paralelização

informáticaufsm
Ciência da Computação

Justificativa

- Rust quer substituir C++
- Desenvolvedores precisam de sugestões e casos de uso
- Paralelismo é essencial para extrair performance

informáticaufsm
Ciência da Computação

Outline

1 Introdução
- Objetivos
- Justificativa

2 Fundamentação
- Rust

3 Desenvolvimento
- Implementação

4 Próximos Passos
- Terminar a port
- Paralelização

informáticaufsm
Ciência da Computação

Rust

- Sistema de Regiões
  - Gerenciamento de memória sem *garbage collection*.
  - *Aliasing* e *thread safety*.
- *Traits*.
- Enums, tuplas e *pattern matching*.

Regiões

- Gerenciamento de memória sem *garbage collection*

```
1  struct Coisa;
2
3  fn foobar() -> Coisa {
4      let a = Coisa;  // Alocado na pilha
5      let b = box Coisa;  // Alocado no heap
6
7      let c = b;  // Movido de b para c
8
9      a  // Move e retorna a
10     // Se última linha não termina com ;, return é implicito
11 }  // c é liberado. a e b já tinham sido movidos
```

informáticaufsm
Ciência da Computação

Regiões

- Gerenciamento de memória sem *garbage collection*

```
1  struct Coisa;
2
3  fn foobar() -> Box<Coisa> {
4      let a = Coisa; // Alocado na pilha
5      let b = box Coisa; // Alocado no heap
6
7      let c = b; // Movido de b para c
8
9      b // Move e tenta retornar b
10 }
```

Regiões

- Gerenciamento de memória sem *garbage collection*

```
<anon>:9:5: 9:6 error: use of moved value: `b`
<anon>:9      b  // Move e tenta retornar b
              ^
<anon>:7:9: 7:10 note: `b` moved here because it has type `Box<
    Coisa>`, which is moved by default (use `ref` to override)
<anon>:7      let c = b;  // Movido de b para c
```

Regiões

- Previne uso não-sincronizado e invalidação de iteradores.

```
1  fn main() {
2      let mut n = 52u32;
3
4      let a = &n;
5      let b = &n;  // OK - Dois ponteiros somente-leitura
6
7      println!("{} {}", *a, *b);
8  }
```

Regiões

- Previne uso não-sincronizado e invalidação de iteradores.

```
1  fn main() {
2      let mut n = 52u32;
3
4      let a = &n;
5      let b = &n;  // OK - Dois ponteiros somente-leitura
6
7      *a = 24;  // ERRO - Não pode modificar ponteiro compartilhado
8
9      println!("{} {}", *a, *b);
10 }
```

```
<anon>:7:5: 7:12 error: cannot assign to immutable dereference
    of `&`-pointer `*a`
<anon>:7     *a = 24;
             ^~~~~~~
```

informáticaufsm
Ciência da Computação

Regiões

- Previne uso não-sincronizado e invalidação de iteradores.

```
1  fn main() {
2      let mut n = 52u32;
3
4      let a = &mut n;  // OK - Um ponteiro mutável
5
6
7      *a = 24;  // OK
8
9      println!("{}", *a);
10 }
```

Regiões

- Previne uso não-sincronizado e invalidação de iteradores.

```
1  fn main() {
2      let mut n = 52u32;
3
4      let a = &mut n;
5      let b = &n;  // ERRO - n já tem uma referência mutável
6      // ...
7  }
```

```
<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;
                         ^
```

informáticaufsm Ciência da Computação

Traits

- Similar a *interfaces* de Java, etc.
  - Trait contém métodos, mas não campos.
  - Tipos que implementam um trait podem ser tratados polimórficamente (durante compilação *e* execução.)
- Implementação do trait para o tipo é separado da definição do tipo.
  - Novos tipos podem implementar traits existentes.
  - Novos traits podem vir com implementações para tipos existentes.
  - Métodos com mesmo nome em traits diferentes podem coexistir no mesmo tipo.

informáticaufsm
Ciência da Computação

Outline

1 Introdução
- Objetivos
- Justificativa

2 Fundamentação
- Rust

3 Desenvolvimento
- Implementação

4 Próximos Passos
- Terminar a port
- Paralelização

informáticaufsm
Ciência da Computação

Visão Geral

- SmallVCM é dividido entre uma parte comum e algoritmos de renderização específicos que utilizam essa infraestrutura.
- A *port* mantém a mesma estrutura de arquivos, tipos e funções, na medida do possível, afim de facilitar comparações.
- Alguns módulos tiveram que ser usar uma estratégia diferente para adaptá-los as funcionalidades e limitações de Rust.

informáticaufsm
Ciência da Computação

Herança

- Rust não tem herança de *structs*.
- Polimorfismo ainda pode ser realizado utilizando *traits*, mas estes não contém dados, apenas métodos.
- Requer divisão de classes base em partes, se estas também conterem campos de dados compartilhados.

Herança

```
1  class AbstractRenderer {
2  public:
3    AbstractRenderer(const Scene& aScene) {/* ... */}
4    virtual ~AbstractRenderer() {}
5    virtual void RunIteration(int aIteration) = 0;
6    void GetFramebuffer(Framebuffer& oFramebuffer) {/* ... */}
7    bool WasUsed() const {/* ... */}
8
9    uint mMaxPathLength;
10   uint mMinPathLength;
11
12 protected:
13   int mIterations;
14   Framebuffer mFramebuffer;
15   const Scene& mScene;
16 }
```

informáticaufsm
Ciência da Computação

Herança

```
1  pub trait AbstractRenderer {
2    fn get_base<'a>(&'a mut self) -> &mut RendererBase<'a>;
3    fn run_iteration(&mut self, iteration: u32);
4  }
5
6  pub struct RendererBase<'a> {
7    pub max_path_length: u32,
8    pub min_path_length: u32,
9
10   pub iterations: u32,
11   pub framebuffer: Framebuffer,
12   pub scene: &'a Scene,
13 }
14
15 impl<'a> RendererBase<'a> {
16   pub fn get_framebuffer(&mut self) -> Framebuffer {/* ... */}
17   pub fn was_used(&self) -> bool {/* ... */}
18 }
```

informáticaufsm
Ciência da Computação

Inicialização de Structs

- Diferentemente de C++, não são permitidos valores não inicializados (sejam variáveis ou campos.)
- Alguns construtores unidos com métodos de inicialização:

```
1  class Framebuffer {
2  public:
3    Framebuffer() {}
4
5    void Setup(const Vec2F& aResolution) {
6      mResolution = aResotluion;
7      mResX = int(aResolution.x);
8      mResY = int(aResolution.y);
9      mColor.resize(mResX * mResY);
10     Clear();
11   }
12   // ...
13 };
```

informáticaufsm
Ciência da Computação

Inicialização de Structs

```
1  pub struct Framebuffer {
2    // ...
3  }
4
5  impl Framebuffer {
6    pub fn setup(resolution: Vec2f) -> Framebuffer {
7      let res_x = resolution.x as uint;
8      let res_y = resolution.y as uint;
9
10     Framebuffer {
11       color: Vec::from_elem(res_x * res_y, vec3s(0.0)),
12       resolution: resolution,
13       res_x: res_x,
14       res_y: res_y,
15     }
16   }
17 }
```

informáticaufsm
Ciência da Computação

Sobrecarga de operadores

- Sobrecarga de operadores é feita implementando-se *traits* especiais.
- `Add`, `Sub`, `Index`, etc.
- Bastante verboso comparado com a sintaxe de C++:

```
1  friend Vec2x<T> operator+(const Vec2x& a, const Vec2x& b) {
2    // ...
3  }
4  // ... x8 operadores, x2 tipos de vetor
```

```
1  impl<T: Num> Add<Vec2x<T>, Vec2x<T>> for Vec2x<T> {
2    #[inline]
3    fn add(&self, o: &Vec2x<T>) -> Vec2x<T> {
4      // ...
5    }
6  }
7  // ... x8 operadores, x2 tipos de vetor
```

- Pode-se usar macros para diminuir a repetição.

Bitflags

- Macro `bitflags!` permite definir flags e bitmasks de maneira *type safe*.

```
1  bitflags! {
2    flags BoxMask: u32 {
3      const LIGHT_CEILING    = 1,
4      const LIGHT_SUN        = 2,
5      const LIGHT_POINT      = 4,
6      const LIGHT_BACKGROUND = 8,
7
8      const LARGE_MIRROR_SPHERE = 16,
9      const LARGE_GLASS_SPHERE  = 32,
10     const SMALL_MIRROR_SPHERE = 64,
11     const SMALL_GLASS_SPHERE  = 128,
12     const GLOSSY_FLOOR = 256,
13     // ...
14   }
15 }
```

Bitflags

- Infelizmente não podem ser usadas como expressões constantes.
- Solução: Voltar a utilizar constantes de inteiros, ou usá-las em situações onde não são necessárias expressões constantes.

```
1 uint g_SceneConfigs[] = {
2   Scene::kBothSmallSpheres  | Scene::kLightSun,
3   Scene::kLargeMirrorSphere | Scene::kLightCeiling,
4   Scene::kBothSmallSpheres  | Scene::kLightPoint,
5   Scene::kBothSmallSpheres  | Scene::kLightBackground
6 };
```

informáticaufsm
Ciência da Computação

Bitflags

- Infelizmente não podem ser usadas como expressões constantes.
- Solução: Voltar a utilizar constantes de inteiros, ou usá-las em situações onde não são necessárias expressões constantes.

```
1  fn get_scene_config(scene_id: uint) -> Option<BoxMask> {
2   match scene_id {
3    1 => Some(scene::BOTH_SMALL_SPHERES | scene::LIGHT_SUN),
4    2 => Some(scene::LARGE_MIRROR_SPHERE| scene::LIGHT_CEILING),
5    3 => Some(scene::BOTH_SMALL_SPHERES | scene::LIGHT_POINT),
6    4 => Some(scene::BOTH_SMALL_SPHERES | scene::LIGHT_BACKGROUND),
7    _ => None
8   }
9  }
```

informáticaufsm
Ciência da Computação

Enums

- Enums não possuem valores inteiros (públicos) associados a elas.
- Teste precisa ser feito utilizando *pattern matching*.

```
1  static const char* GetName(Algorithm aAlgorithm)
2  {
3    static const char* algorithmNames[7] =
4    {
5      "eye light", "path tracing", "light tracing",
6      "progressive photon mapping",
7      "bidirectional photon mapping",
8      "bidirectional path tracing",
9      "vertex connection and merging"
10   };
11
12   if(aAlgorithm < 0 || aAlgorithm > 7)
13     return "unknown algorithm";
14
15   return algorithmNames[aAlgorithm];
16 }
```

informáticaufsm
Ciência da Computação

Enums

- Enums não possuem valores inteiros (públicos) associados a elas.
- Teste precisa ser feito utilizando *pattern matching*.

```
1  fn get_name(self) -> &'static str {
2    match self {
3      EyeLight => "eye light",
4      PathTracing => "path tracing",
5      LightTracing => "light tracing",
6      ProgressivePhotonMapping => "progressive photon mapping",
7      BidirectionalPhotonMapping => "bidirectional photon mapping",
8      BidirectionalPathTracing => "bidirectional path tracing",
9      VertexConnectionMerging => "vertex connection and merging",
10   }
11 }
```

informáticaufsm
Ciência da Computação

Outline

1 Introdução
- Objetivos
- Justificativa

2 Fundamentação
- Rust

3 Desenvolvimento
- Implementação

4 Próximos Passos
- Terminar a port
- Paralelização

informáticaufsm
Ciência da Computação

Terminar a *port*

- O SmallVCM continuará sendo portado até um ponto em que esteja funcional e consiga gerar imagens similares ao original.
  - Não é necessário portar todos os algoritmos de renderização.
  - Após a finalização do resto do trabalho, será portado o restante do código, se houver tempo disponível.
- Serão feitas comparações de performance com o original, afim de identificar regressões de performance e suas causas.

informáticaufsm
Ciência da Computação

Outline

1 Introdução
- Objetivos
- Justificativa

2 Fundamentação
- Rust

3 Desenvolvimento
- Implementação

4 Próximos Passos
- Terminar a port
- Paralelização

informáticaufsm
Ciência da Computação

Paralelização

- Serão exploradas técnicas de paralelização e como podem ser integradas na versão Rust. Algumas das modalidades a se investigar são:
  - Multi-threading
  - SIMD
  - GPU Computing

informáticaufsm
Ciência da Computação

Paralelização

- Multi-threading
  - Trabalho de renderização divisível em sub-regiões menores.
  - Investigar como o uso de OpenMP do código original pode ser mapeado para a biblioteca padrão de Rust.

Paralelização

- SIMD
  - O *rustc* tem suporte básico a SIMD: Existem tipos que representam um vetor de tamanho fixo, suportando operações aritméticas.
  - Não existe suporte a operações mais complexas de *shuffling*, comparações ou *masking*.
  - Investigar como pode ser usado no renderizador e se será necessário adicionar melhor suporte a estas operações no compilador ou biblioteca padrão.

Paralelização

- GPU Computing
  - Existe um projeto *proof of concept* que permite compilar código Rust para CUDA e executá-lo na GPU.
  - Seria necessário determinar se continua utilizável com novas versões do compilador e se parte do código do renderizador pode ser adaptado para ser acelerado desta maneira.

informáticaufsm
Ciência da Computação

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

Yuri Kunde Schlesner
Orientador: Profª Drª Andrea Scwertner Charão

Ciência da Computação
Universidade Federal de Santa Maria

20/10/2014

informáticaufsm
Ciência da Computação