Raytracing - Rust Proseminar

Sami Shalayel, Daniel Freiermuth, Carl Schwan

Projektstruktur : Interceptable Trait

Interceptable Trait

Wir können alles rendern, was den „Interceptable”-Trait implementiert :

Projektstruktur : Interceptable Trait

Interceptable Trait

Wir können alles rendern, was den „Interceptable”-Trait implementiert :
Dieser gibt an, ob, wo und wie das Objekt von einem Lichtstrahl getroffen wird.

Projektstruktur : Interceptable Trait

Interceptable Trait

Wir können alles rendern, was den „Interceptable”-Trait implementiert :
Dieser gibt an, ob, wo und wie das Objekt von einem Lichtstrahl getroffen wird.
Beispiele für „Interceptable”:

- Kugeln

Projektstruktur : Interceptable Trait

Interceptable Trait

Wir können alles rendern, was den „Interceptable”-Trait implementiert :
Dieser gibt an, ob, wo und wie das Objekt von einem Lichtstrahl getroffen wird.
Beispiele für „Interceptable”:

- Kugeln
- Ebenen

Projektstruktur : Interceptable Trait

Interceptable Trait

Wir können alles rendern, was den „Interceptable”-Trait implementiert :
Dieser gibt an, ob, wo und wie das Objekt von einem Lichtstrahl getroffen wird.
Beispiele für „Interceptable”:

- Kugeln
- Ebenen
- Dreiecke

Projektstruktur : Interceptable Trait

Interceptable Trait

Wir können alles rendern, was den „Interceptable”-Trait implementiert :
Dieser gibt an, ob, wo und wie das Objekt von einem Lichtstrahl getroffen wird.
Beispiele für „Interceptable”:

- Kugeln
- Ebenen
- Dreiecke
- Beschleunigungsstrukturen

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.
Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.
Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe
- Diffuse shader : Diffuses Licht

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.
Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe
- Diffuse shader : Diffuses Licht
- Mirror shader : Spiegelt Licht zurück

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.

Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe
- Diffuse shader : Diffuses Licht
- Mirror shader : Spiegelt Licht zurück
- Specular shader : Glanzlichter

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.

Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe
- Diffuse shader : Diffuses Licht
- Mirror shader : Spiegelt Licht zurück
- Specular shader : Glanzlichter

Und können sie kombinieren :

- Additive shader (für $+$)

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.

Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe
- Diffuse shader : Diffuses Licht
- Mirror shader : Spiegelt Licht zurück
- Specular shader : Glanzlichter

Und können sie kombinieren :

- Additive shader (für $+$)
- Multiplicative shader (für *)

Projektstruktur : Shader Trait

Shader Trait

Jeder Shader liefert eine Farbe für einen Lichtstrahl-Objekt-Schnitt.

Wir haben folgende Shader implementiert :

- Monochrome shader : Nur eine Farbe
- Diffuse shader : Diffuses Licht
- Mirror shader : Spiegelt Licht zurück
- Specular shader : Glanzlichter

Und können sie kombinieren :

- Additive shader (für $+$)
- Multiplicative shader (für *)
- Chess shader : 2 abwechselnde Shader

Shader Trait : Phong Shader

Phong Shader

Die std::ops::Add und std::ops::Mul Traits für `Box<Shader>` erleichtern das Bauen des Phong-Shader :

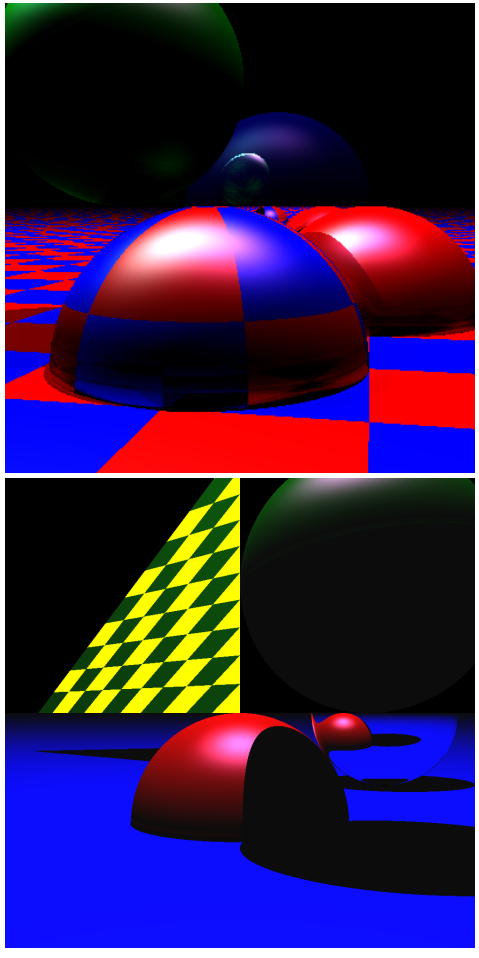
Shader Trait : Phong Shader

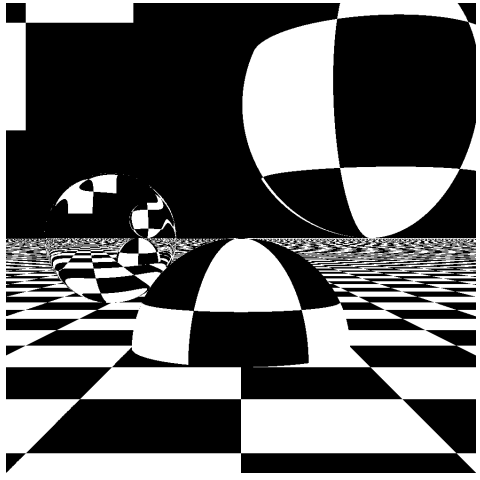
Phong Shader

Die std::ops::Add und std::ops::Mul Traits für `Box<Shader>` erleichtern das Bauen des Phong-Shader :

```
pub fn get_phong(color: Vector3<f64>) -> Box<Shader> {
    let diffuse_shader = DiffuseShader::new(color);
    let specular_shader = SpecularShader::new(10.0);
    let ambient_shader = AmbientShader::new(color);
    return 0.5 * diffuse_shader
        + specular_shader
        + 0.8 * ambient_shader;
}
```

Beispiel





Projektstruktur : Camera Trait & Wavefront Parser

Camera Trait

```
pub trait Camera {
    fn render(&self, world: &World) -> DynamicImage;
}
```

Projektstruktur : Camera Trait & Wavefront Parser

Camera Trait

```
pub trait Camera {
    fn render(&self, world: &World) -> DynamicImage;
}
```

- Equilinear Camera : „Normale" Kamera
- Equirectangular Camera : „360 Grad" Kamera

Projektstruktur : Camera Trait & Wavefront Parser

Camera Trait

```
pub trait Camera {
    fn render(&self, world: &World) -> DynamicImage;
}
```

- Equilinear Camera : „Normale" Kamera
- Equirectangular Camera : „360 Grad" Kamera

Wavefront Parser

Projektstruktur : Camera Trait & Wavefront Parser

Camera Trait

```
pub trait Camera {
    fn render(&self, world: &World) -> DynamicImage;
}
```

- Equilinear Camera : „Normale" Kamera
- Equirectangular Camera : „360 Grad" Kamera

Wavefront Parser

3D Objekte werden als Obj-Wavefront format eingelesen und geparst.

Projektstruktur : Camera Trait & Wavefront Parser

Camera Trait

```
pub trait Camera {
    fn render(&self, world: &World) -> DynamicImage;
}
```

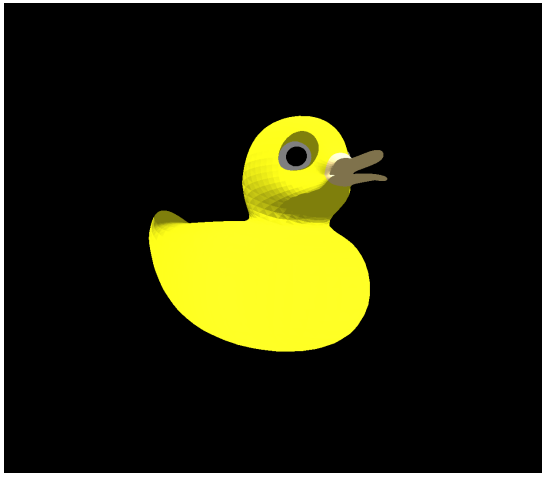
- Equilinear Camera : „Normale" Kamera
- Equirectangular Camera : „360 Grad" Kamera

Wavefront Parser

3D Objekte werden als Obj-Wavefront format eingelesen und geparst.
Dafür haben wir den wavefront_obj crate leicht verändert.

Projektstruktur : Beispiel

Beispiel



Benutzte Rustfeatures : Operator Überladungen & Dynamic Dispatch

Um 2 Shader zu addieren:

Benutzte Rustfeatures : Operator Überladungen & Dynamic Dispatch

Um 2 Shader zu addieren:

```
impl Add for Box<Shader> {
    type Output = Box<Shader>;
    fn add(self, other: Box<Shader>) -> Box<Shader> {
        Box::new(AdditiveShader {
            shader1: self,
            shader2: other,
        })
    }
}
```

Benutzte Rustfeatures : Nalgebra und std::f64

Nalgebra

Nalgebra ist eine Algebra-Bibliothek für Rust, die (fast) alles kann und die die Berechnung mit Vektoren vereinfacht :

Benutzte Rustfeatures : Nalgebra und std::f64

Nalgebra

Nalgebra ist eine Algebra-Bibliothek für Rust, die (fast) alles kann und die die Berechnung mit Vektoren vereinfacht :

- Operatoren sind für Vektoren/Matrizen überladen

Benutzte Rustfeatures : Nalgebra und std::f64

Nalgebra

Nalgebra ist eine Algebra-Bibliothek für Rust, die (fast) alles kann und die die Berechnung mit Vektoren vereinfacht :

- Operatoren sind für Vektoren/Matrizen überladen
- Interessante Hierarchie Struktur mit Generics :

```
type Vector3<N> = VectorN<N, U3>;
type VectorN<N, D> = MatrixMN<N, D, U1>;
type MatrixMN<N, R, C> =
  Matrix<N, R, C, Owned<N, R, C>>;
```

Benutzte Rustfeatures : Nalgebra und std::f64

Nalgebra

Nalgebra ist eine Algebra-Bibliothek für Rust, die (fast) alles kann und die die Berechnung mit Vektoren vereinfacht :

- Operatoren sind für Vektoren/Matrizen überladen
- Interessante Hierarchie Struktur mit Generics :

```
type Vector3<N> = VectorN<N, U3>;
type VectorN<N, D> = MatrixMN<N, D, U1>;
type MatrixMN<N, R, C> =
  Matrix<N, R, C, Owned<N, R, C>>;
```

std::f64

Alles was man braucht um mit floats zu arbeiten :

- Round, Log, Exp, Abs, ...

Benutzte Rustfeatures : Nalgebra und std::f64

Nalgebra

Nalgebra ist eine Algebra-Bibliothek für Rust, die (fast) alles kann und die die Berechnung mit Vektoren vereinfacht :

- Operatoren sind für Vektoren/Matrizen überladen
- Interessante Hierarchie Struktur mit Generics :

```
type Vector3<N> = VectorN<N, U3>;
type VectorN<N, D> = MatrixMN<N, D, U1>;
type MatrixMN<N, R, C> =
  Matrix<N, R, C, Owned<N, R, C>>;
```

std::f64

Alles was man braucht um mit floats zu arbeiten :

- Round, Log, Exp, Abs, ...
- aber auch Trigonometrische Funktionen wie cos, sin, tanh, ...

```
fn main() {
    println!("Hello {}!", 3.14f64.cos());
}
```

Benutzte Rustfeatures : Error Handling

In Rust wird oft `Result<T, Err>` benutzt.

- Sogar die Main kann einen `Result<T, Err>` zurückgeben

Benutzte Rustfeatures : Error Handling

In Rust wird oft `Result<T, Err>` benutzt.

- Sogar die Main kann einen `Result<T, Err>` zurückgeben
- `try!(...)` bzw `?` : ersetzt ein Result durch ihren Wert oder returned den Error

Benutzte Rustfeatures : Error Handling

In Rust wird oft `Result<T, Err>` benutzt.

- Sogar die Main kann einen `Result<T, Err>` zurückgeben
- `try!(...)` bzw `?` : ersetzt ein Result durch ihren Wert oder returned den Error
- Eigener Fehlerenum, dass die eigenen library-Fehler wrappt

Rust Downsides

- erzwungene einheitliche Pointer : kein Vertauschen zwischen `Box<T>` und `&T` möglich

  Zum Beispiel: Um eine Box zu teilen müsste man mit `&Box<T>` arbeiten

Rust Downsides

- erzwungene einheitliche Pointer : kein Vertauschen zwischen `Box<T>` und `&T` möglich

  Zum Beispiel: Um eine Box zu teilen müsste man mit `&Box<T>` arbeiten

- Rayon : eine Library um Iteratoren zu parallelisieren

  Kompilierfehler und Dokumentation haben sich wiedersprochen

Rust Downsides

- erzwungene einheitliche Pointer : kein Vertauschen zwischen `Box<T>` und `&T` möglich

  Zum Beispiel: Um eine Box zu teilen müsste man mit `&Box<T>` arbeiten

- Rayon : eine Library um Iteratoren zu parallelisieren

  Kompilierfehler und Dokumentation haben sich wiedersprochen

- `cargo bench` : nur als nightly, kann nur Untercrates benchen

Rust Downsides

- erzwungene einheitliche Pointer : kein Vertauschen zwischen `Box<T>` und `&T` möglich

  Zum Beispiel: Um eine Box zu teilen müsste man mit `&Box<T>` arbeiten

- Rayon : eine Library um Iteratoren zu parallelisieren

  Kompilierfehler und Dokumentation haben sich wiedersprochen

- `cargo bench` : nur als nightly, kann nur Untercrates benchen

- Keine „Down-grades" von Trait zu Super-Traits möglich : man muss From und Into implementieren

Lessons learned

- Wenn man Referenzen in Structs benutzt, braucht man Lifetimes
- Serde
- Um Felder von Structs moven zu können, muss man zuerst den struct zerstören

Fragen?

TODO Video