Project: Voxel-Renderer

Door Lars Jensen

Inhoud

[Gebuikte Technologieën 2](#_Toc116651137)

[Window en Mainloop 3](#_Toc116651138)

[Rendering 4](#_Toc116651139)

[De UI 5](#_Toc116651140)

[De voxel shader 6](#_Toc116651141)

[Wat is een shader? 6](#_Toc116651142)

[CPU-side 6](#_Toc116651143)

[Raymarching 8](#_Toc116651144)

[Wat is raymarching? 8](#_Toc116651145)

Inleiding

Voor mijn oriëntatie project heb ik een voxel renderer geschreven. Deze neemt een array van posities en kleuren en zorgt hiervoor dat het getekend wordt als een aantal 3d kubussen op een grid. Dit soort renderers worden vaak gebruikt voor interactieve werelden en simulaties visualiseren met cellular-automata. Voor dit project ga ik alleen de voxels tekenen met een algoritme genaamd raymarching.

# Gebuikte Technologieën

Talen: Rust, WGSL

Graphics Backend: WGPU

Chart, funnel chart

Description automatically generated

Uitwerking

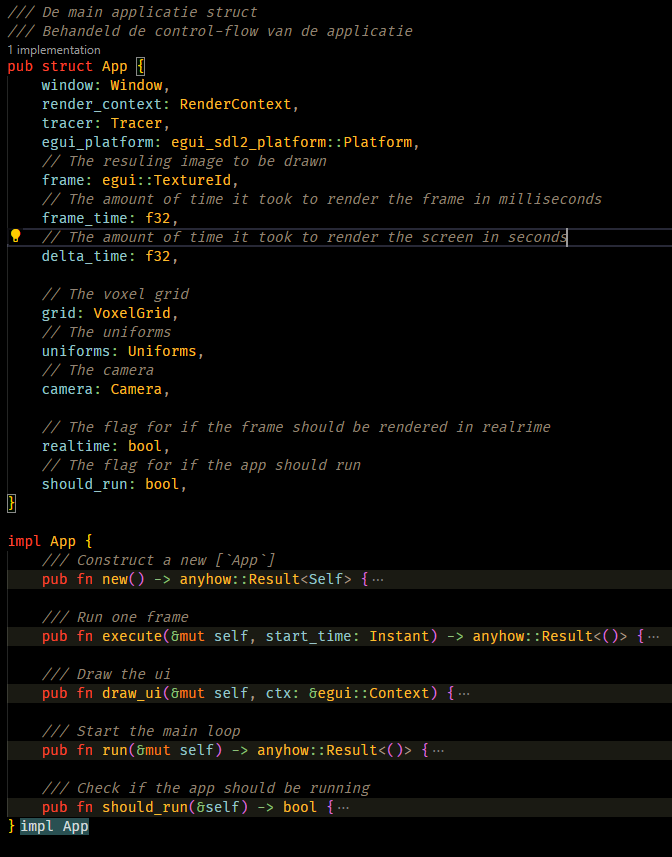
# Window en Mainloop

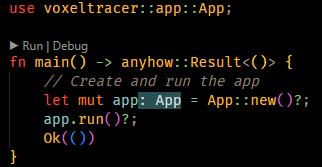
Om een renderer te schrijven moet ik eerst een venster hebben om het resultaat op te presenteren.  
Hiervoor heb ik een venster opgesteld met een library genaamd Sdl2.

Sdl2 is een library die voor verschillende platforms een venster kan creëren, hiermee kan ik ook makkelijk een venster creëren waar mijn programma op kan tekenen.

Sdl2 heeft het nodig dat de events van het venster behandeld voor elke frame, hiervoor heb ik een simpele mainloop moeten programmeren. Uiteindelijk heb ik dit structuur in een aantal structs ingepakt zodat het makkelijker is te gebruiken.

Text

Description automatically generatedHet resultaat is een `App` struct en een `Window` struct:

Zo is alles makkelijk te creëren in de entrypoint:  


# Rendering

Om een fatsoenlijke renderer te hebben moet ik de GPU kunnen gebruiken om de frame te tekenen en de berekeningen te kunnen doen. Om dit makkelijker te maken moet je gebruik maken van een rendering-backend.

Er zijn verschillende opties om uit te kiezen, zoals OpenGL, Vulkan, DirectX, Metal etc. Welke backend het beste is ligt er aan welke OS de user gebruikt. Hiervoor komt WGPU aan bod, WGPU is een library die de api’s van deze backends abstract en dat op runtime kiest voor welke het beste is gebaseerd op de device waarom de app gerund wordt. De api en logica van WGPU is gebasseerd op de WebGPU API-spec waardoor de docs en voorbeelden van beide api’s van toepassing kunnen zijn.

De logica voor de renderer heb ik gestopt in een `RenderContext` struct:

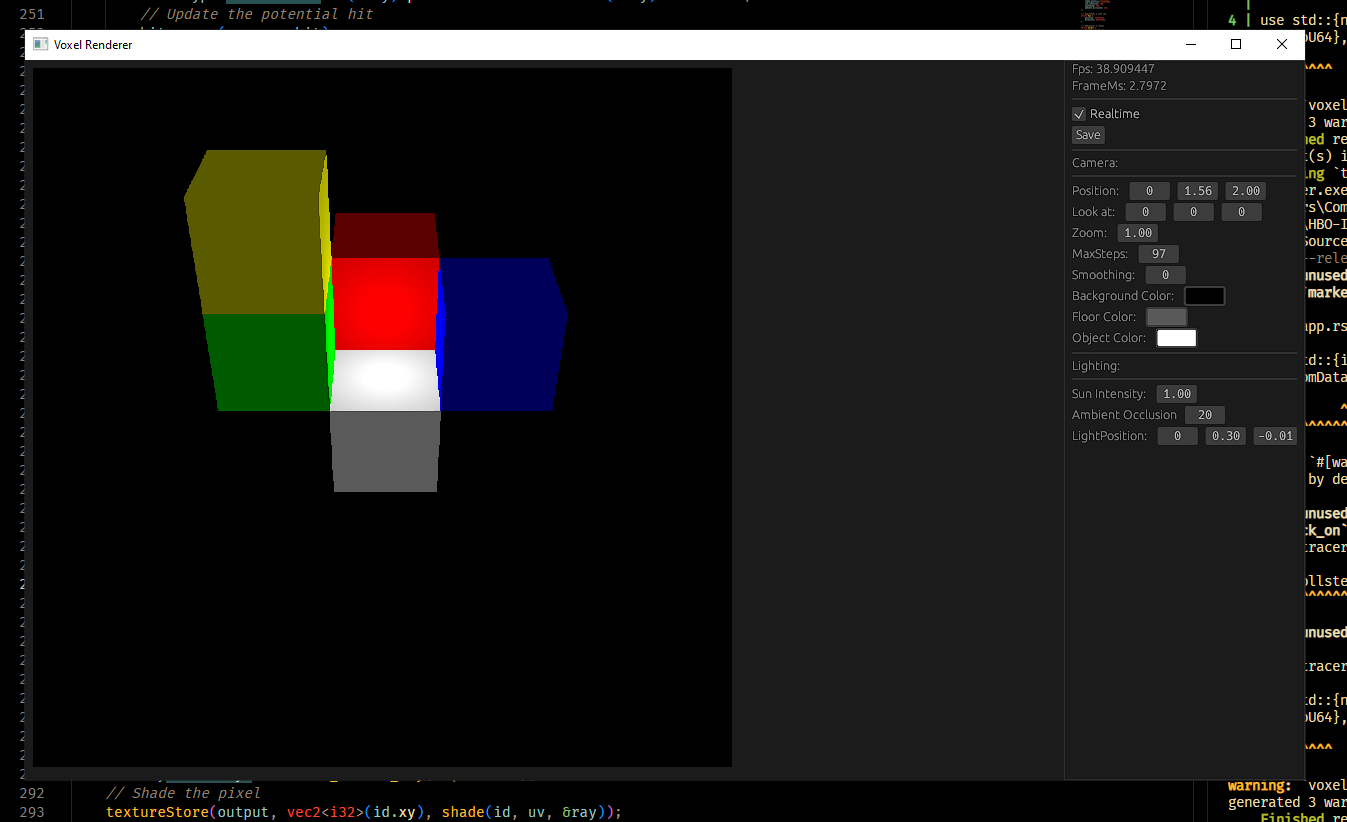
Deze struct zorgt ervoor dat er een frame tot een zogenaamde ‘surface texture’ getekend wordt en dat deze op het venster getoond wordt.

# De UI

De ui is vooral bedoeld om het resultaat te presenteren op het scherm, maar ook om wat van de settings te veranderen. Om het maken van ui makkenlijker te maken heb ik de `egui` library gebruikt.

Egui is een library waarmee je UI kan tekenen voor verschillende renderering-backends. Het is op zo’n manier geschreven dat het makkelijk is te gebruiken en integreren in verschillende renderer’s/engines.

Egui heeft wel wat meer setup nodig om met mijn renderer te werken, dit was makkenlijk op te zetten met de `egui\_sdl2\_platform` library (die ik zelf geschreven heb) en de `egui\_wgpu\_backend` library.

Het eindresultaat is een ui met een viewport:  


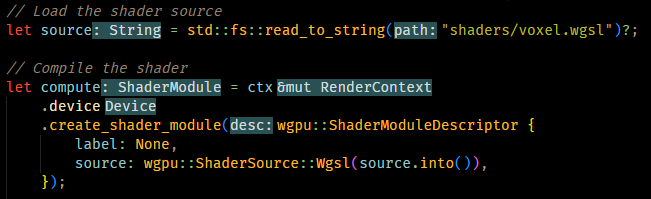
# De voxel shader

## Wat is een shader?

Een shader is een programma die gerund kan worden op de GPU. Hiermee is het mogelijk om de kleur van meerdere pixels in parallel te berekenen. Er zijn verschillende soorten shaders maar degene die de renderer gebruikt is een compute-shader vanwegen het feit dat je hiermee meer controle hebt over het aantal pixels dat de GPU in parallel tekent (ik heb voor dit project 8 per keer gekozen sinds dit het standaard is voor textures).

## CPU-side

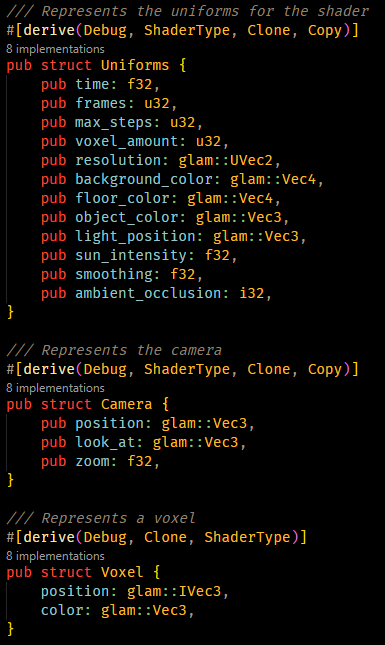
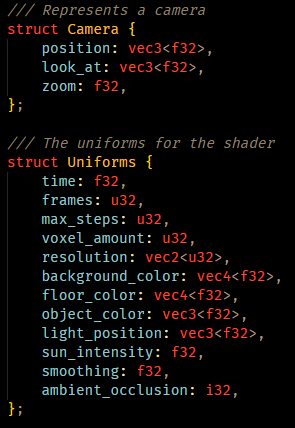
Deze shader krijgt een texture toegestuurd waarop het de veranderingen kan maken, en de grid van voxels die het moet tekenen. Ook krijgt het wat data toegestuurd zodat het wat context heeft voor de app.

De source code van de shader zit in een aparte file en is geschreven in een speciale shader-taal genaamd wgsl. De shader wordt op runtime compiled door WGPU. Dit kan makkenlijk gedaan worden door de file te lezen als string:  


De logica hiervoor heb ik ingepakt in een `Tracer` struct die het sturen van de data en het uitvoeren van de shader behandeld:  


Als de shader uitgevoerd wordt wordt de texture id ge-returned zodat de UI het kan tekenen op het scherm.

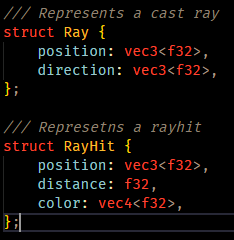
De data van de shaders heb ik in aparte structs gezet, ik heb ze de `ShaderType` type gegeven zodat ik ze later de goede memory alignement kan geven van 16 met gebruik van de `encase` library. Deze structs moesten worden nagemaakt in de shader code:

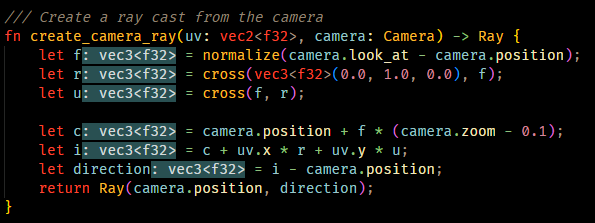


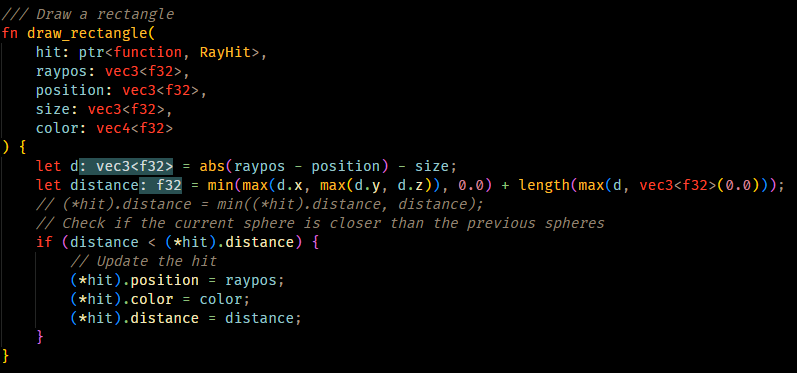
# Raymarching

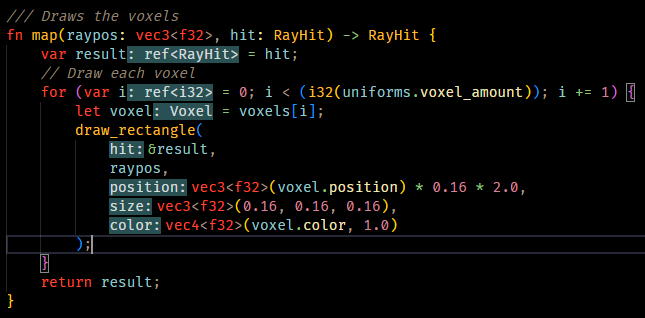
## Wat is raymarching?

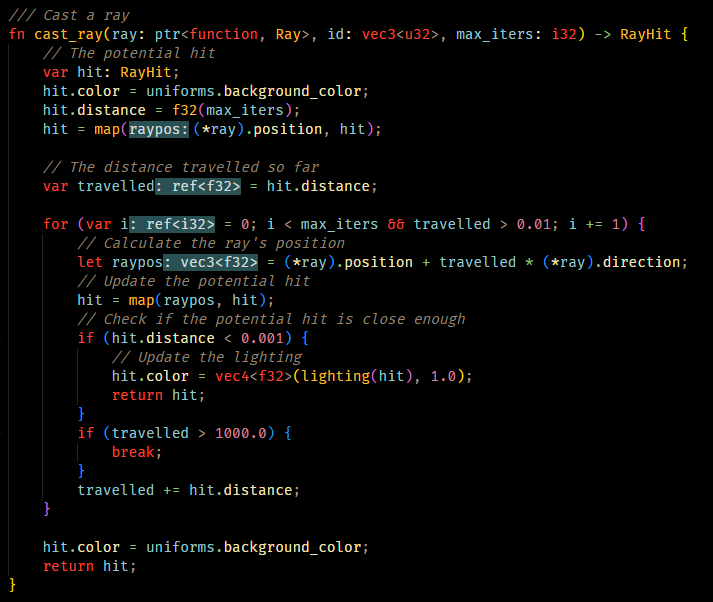
Raymarching is een techniek van vormen tekenen op basis van de afstand tot de camera. Dit doet het door een ray in stappen te casten vanuit de camera, en pas iets tekenen als de stappen te klein worden. Het bepaald hoe groot de stappen moet zijn door de afstand tot het dichtste object te berekenen en de stap even groot te maken. Zo kun je verschillende vormen tekenen door alleen maar de afstand-functie in te voeren.

In de shader heb ik een paar structs gemaakt die een ray en een rayhitpunt representeren:  


Ook heb ik een functie gemaakt die een ray maakt vanuit een bepaald punt van de camera:  


Verschillende vormen hebben verschillende functies voor afstanden, de afstandfunctie voor een cubus (voxel) is als volgt:  


Om de wereld te kunnen tekenen heb ik een `map` functie geschreven die de hitpoint returns van het dichste object:  


De kleurberekening wordt gedaan door het raymarch-algoritme te gebruiken om de kleur van het geraakte punt te pakken en wat licht-berekeningen te doen:  


Uiteindelijk zet het de pixel naar de geresulteerde kleur:  
