

Bachelorarbeit
Studiengang Medieninformatik

WebGPU

von

Laurin Agostini

60526

Betreuender Professor: Prof. Dr. Winfried Bantel
Zweitprüfer: Prof. Dr. Carsten Lecon

Einreichungsdatum: XX. Juni 2020

Eidesstattliche Erklärung

Hiermit erkläre ich, **Laurin Agostini**, dass ich die vorliegenden Angaben in dieser Arbeit wahrheitsgetreu und selbständig verfasst habe.

Weiterhin versichere ich, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, dass alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ort, Datum

Unterschrift (Student)

Kurzfassung

In dieser Bachelorarbeit geht es um die neuartige Grafik-API WebGPU, die einen Nachfolger zu WebGL darstellt. Mit WebGPU ist es möglich, detaillierte 3D-Szenen, aufwendige Simulationen und XXX in Echtzeit direkt im Webbrowser zu berechnen und darzustellen.

Inhaltsverzeichnis

Eidesstattliche Erklärung	i
Kurzfassung	ii
Inhaltsverzeichnis	iii
Abbildungsverzeichnis	iv
Tabellenverzeichnis	v
Listings	vi
Abkürzungsverzeichnis	viii
1. Einleitung	1
1.1. Motivation	1
1.2. Problemstellung und -abgrenzung	1
1.3. Ziel der Arbeit	1
1.4. Vorgehen	1
2. Grundlagen	2
2.1. GPU	2
2.1.1. Aufgaben und Unterschied zur CPU	2
2.1.2. Integrierte und dedizierte GPUs	3
2.2. Rendering-Pipeline [Vgl. 2, The Graphics Rendering Pipeline]	4
2.2.1. Architektur	4
2.2.2. Anwendung	5
2.2.3. Geometrieverarbeitung	5
2.2.4. Rasterung	8
2.2.5. Pixelverarbeitung	11
2.3. Grafik-APIs [Vgl. 2, The Evolution of Programmable Shading and APIs]	12
3. WebGPU	15
3.1. Entstehung [Vgl. 46]	15
3.2. Beschreibung	16
3.3. WGSL	16
3.3.1. Übersicht	16

3.3.2. Kritik an WGSL	16
3.3.3. Datentypen	17
3.4. Der aktuelle Stand	20
3.4.1. Spezifikation	20
3.4.2. Implementierung	20
3.5. Nutzen außerhalb von Grafik	21
4. Implementierung einer WebGPU-Applikation	22
4.1. spider	22
4.1.1. Überblick	22
4.1.2. Codestyle	24
4.2. emscripten [5]	28
4.2.1. Funktionsweise	28
4.3. Benutzte WebGPU-Features	29
4.4. Besonderheiten bei der Entwicklung einer GPU Applikation	29
5. Evaluierung	33
6. Zusammenfassung und Ausblick	34
6.1. Erreichte Ergebnisse	34
6.2. Ausblick	34
6.2.1. Erweiterbarkeit der Ergebnisse	34
6.2.2. Übertragbarkeit der Ergebnisse	34
Referenzen	34
A. Anhang A	39
B. Anhang B	43

Abbildungsverzeichnis

2.1. Beispiel einer Grafikkarte (Zotax Gaming 2080 ti) [40]	3
2.2. Grundsätzlicher Aufbau der Rendering-Pipeline, unterteilt in die vier Abschnitte <i>Anwendung</i> , <i>Geometrieverarbeitung</i> , <i>Rasterung</i> und <i>Pixelverarbeitung</i>	4
2.3. Transformation von Objekt bezogenen Vertexdaten in den <i>world space</i> , dann in den <i>view space</i> und schlussendlich in den <i>clip space</i> . Die finale Umwandlung vom <i>clip space</i> in den <i>screen space</i> passiert dabei automatisch und verschiebt den Ursprung von der Mitte in entweder die ober oder untere (abhängig von der verwendeten Grafik-API) linke Ecke. Quelle: https://learnopengl.com/img/getting-started/coordinate_systems.png [22]	6
2.4. Zwei Dreiecke T1 und T2 im <i>screen space</i> . T1 wird hierbei komplett gerastert und T2 erst so zerteilt, dass nur der sichtbare Bereich gerastert werden muss.	9
2.5. Interpolation von Vertexdaten innerhalb eines Dreiecks am Beispiel der Farbinformation der Vertices mithilfe von baryzentrischen Koordinaten [35]	10
2.6. Zeitlinie mit wichtigen Grafik-API- und Hardwareerscheinungen [2, S. 38]	12
4.1. Interaktive 3D-Szene mit UserInterface. Bildschirmaufnahme durch Verfasser.	24
4.2. Übersicht der Optionen zum Starten von <i>Google Chrome Canary</i> in <i>Microsoft PIX on Windows</i> [26]. Bildschirmaufnahme durch Verfasser.	31
4.3. Anzeige nach der Erfassung eines Einzelbildes in <i>Microsoft PIX on Windows</i> [26]. Bildschirmaufnahme durch Verfasser.	32

Tabellenverzeichnis

2.1. Vergleich aktueller CPUs und GPUs	2
3.1. Skalare Typen in WGSL mit den äquivalenten Typen in GLSL und HLSL	17
4.1. Verwendete Prefixe	25

Listings

2.1. Beispiel eines Vertex Shaders in OpenGL Shading Language (GLSL), welcher so in der spider -Engine verwendet wird	8
3.1. „Swizzling“ von Komponenten in GLSL	20
4.1. Grundgerüst einer Applikation mit der spider -Engine	23
4.2. Definition von Strukturen	25
4.3. Beispiele zu bestimmten Initialisierern (Unter Verwendung der in Listing 4.2 definierten Struktur <i>MyStructure</i>)	26
4.4. Verwendung einer <i>Desc</i> -Struktur zum Übergeben von Argumenten an eine Funktion	27
4.5. Verwendung von <i>handles</i>	28
A.1. Kompletter C99-Quellcode zur Erstellung einer interaktiven 3D-Szene mit der spider -Engine	39
B.1. Kompletter GLSL-Code des Fragment-Shaders der in der spider - Engine für das Physically Based Rendering (PBR) benutzt wird	43

Abkürzungsverzeichnis

CPU	Central Processing Unit (dt. Hauptprozessor).....	2
GPU	Graphics Processing Unit (dt. Grafikprozessor)	2
SMT	Simultaneous Multithreading	2
PCIe	Peripheral Component Interconnect Express.....	3
API	Application Programming Interface (dt. Programmierschnittstelle)	20
WSL	Windows Subsystem for Linux	22
GLSL	OpenGL Shading Language.....	vii
HLSL	High Level Shading Language	13
PBR	Physically Based Rendering.....	vii
AMD	Advanced Micro Devices, Inc	13

1. Einleitung

1.1. Motivation

1.2. Problemstellung und -abgrenzung

1.3. Ziel der Arbeit

1.4. Vorgehen

2. Grundlagen

2.1. GPU

2.1.1. Aufgaben und Unterschied zur CPU

Wie der Name, **Grafik**prozessor oder **graphics processing unit**, schon andeutet, ist der Primärzweck einer Graphics Processing Unit (dt. Grafikprozessor) (GPU) die Erstellung und Manipulation von Bildinhalten und anschließender Ausgabe dieser an ein Anzeigegerät. Während eine Central Processing Unit (dt. Hauptprozessor) (CPU) generell darauf ausgelegt ist, ein weites Spektrum von Aufgaben sequenziell bearbeiten zu können, ist eine GPU darauf spezialisiert möglichst viele einfache Aufgaben, insbesondere Fließkommaoperationen, parallel zu bearbeiten. Diesen Unterschied sieht man direkt, wenn man die Leistungsdaten aktueller CPUs und GPUs vergleicht:

Hier sieht man sofort, dass GPUs eine um mehrere Größenordnungen höhere Anzahl von Kernen haben, welche allerdings mit einer niedrigeren Taktrate laufen. Zwar konnte in den letzten Jahren ein starker Anstieg von Prozessorkernen in Endbenutzer-CPU's beobachtet werden, doch auch eine explizite Workstation-CPU, wie ein AMD Ryzen™ Threadripper™ 3990X [1] mit 64 physischen und durch Simultaneous Multithreading (SMT) 128 logischen Prozessorkernen ist weit von den Kernzahlen einer aktuellen Einstiegs-GPU entfernt. Wie schon angemerkt, kann man die Prozessorkerne von CPUs und GPUs nicht direkt vergleichen, jedoch zei-

Name	Anzahl Kerne	Basis-/Turbotakt (MHz)
High-end		
Intel® Core™ i9-9900K [16]	8	3.600 / 5.000
NVIDIA GeForce RTX 2080Ti [31]	4.352	1.350 / 1.545
Low-end		
Intel® Core™ i3-9100 [15]	4	3.600 / 4.200
NVIDIA GeForce GTX 1650 [30]	896	1.485 / 1.665

Legende: CPU und GPU

Tabelle 2.1.: Vergleich aktueller CPUs und GPUs



Abbildung 2.1.: Beispiel einer Grafikkarte (Zotax Gaming 2080 ti) [40]

gen sie deutlich die Spezialisierung der GPUs auf parallele Verarbeitung mit einem erhöhtem Datendurchsatz.

Zu den klassischen Aufgaben einer GPU gehören Grafik-, Rechen-, Medien- und Displayfunktionalitäten. Durch die fortlaufende Entwicklung von fest in die Hardware programmierte Abläufe zu frei programmierbaren Anwendungen (ähnlich zu einer CPU), werden jedoch immer mehr Anwendungen möglich, welche als **GPUCompute** zusammengefasst werden. Dies führte auch dazu, dass man in den letzten Jahren einen enormen Anstieg von GPUs in Rechenzentren zum Beschleunigen von Anwendungen wie zum Beispiel *machine learning* oder *crypto mining*, beobachten konnte. Da sich WebGPU jedoch eher auf den klassischen Aufgabenbereich der GPU bezieht, und dafür eine moderne Schnittstelle bereitstellt, wird sich im folgenden primär auf diese bezogen.

2.1.2. Integrierte und dedizierte GPUs

Mittlerweile haben die meisten Endbenutzer-CPU's eine integrierte GPU (oft auch **iGPU** genannt) um die üblichen Multimediaaufgaben zu beschleunigen und Last von der CPU zu nehmen. Integrierte GPUs sind aber für aufwendige Berechnungen wie zum Beispiel für grafikintensive Anwendungen nicht ausreichend. Dafür werden dedizierte Grafikkarten benötigt, welche der GPU dedizierten (daher der Name) Arbeitsspeicher und oft auch eine aktive Kühlung bereitstellen. Grafikkarten werden heutzutage meist über Peripheral Component Interconnect Express (PCIe) angeschlossen.

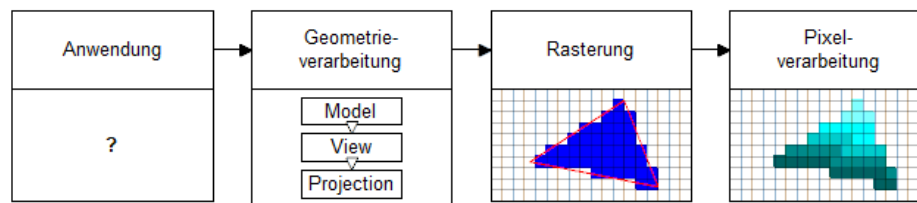


Abbildung 2.2.: Grundsätzlicher Aufbau der Rendering-Pipeline, unterteilt in die vier Abschnitte *Anwendung*, *Geometrieverarbeitung*, *Rasterung* und *Pixelverarbeitung*

2.2. Rendering-Pipeline (Vgl. 2, The Graphics Rendering Pipeline)

The main function of the [graphics rendering] pipeline is to generate, or *render*, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, and more. The rendering pipeline is thus the underlying tool for real-time rendering.

– Real-Time Rendering [2, S. 11]

2.2.1. Architektur

Grundsätzlich besteht die Rendering-Pipeline aus vier Abschnitten, die mehr oder weniger frei programmierbar sind (siehe Abbildung 2.2). Generell läuft der Abschnitt *Anwendung* auf der CPU (kann aber auch teilweise mithilfe von GPU-Compute auf der GPU implementiert werden) und beschreibt die Logik der Anwendung. Die drei folgenden Abschnitte *Geometrieverarbeitung*, *Rasterung* und *Pixelverarbeitung* laufen alle auf der GPU, wobei die *Geometrie*- und *Pixelverarbeitung* hier frei mit *Shadern* programmierbar sind und die *Rasterung* nur über Parameter konfiguriert werden kann.

Ursprünglich wurde der Begriff *Shader* für Programme benutzt, die zur Farb- und Helligkeitsbestimmung einer 3D-Szene beigetragen haben. Da diese Aufgaben sich dann als Spezialität der GPU herausgestellt haben, wurde der Begriff *Shader* immer weiter gefasst und bezeichnet heute grundsätzlich ein GPU-Programm.

2.2.2. Anwendung

Der Abschnitt *Anwendung* kann nicht konkret beschrieben werden, da er sich, wie der Name schon sagt, von Anwendung zu Anwendung unterscheidet. Jedoch kann man grundsätzlich sagen, dass hier die Logik der Anwendung stattfindet, wie zum Beispiel:

- Benutzereingaben verarbeiten und auswerten
- Physikberechnungen zwischen den Objekten
- Ressourcen von der Festplatte laden

Damit bereitet der Abschnitt *Anwendung* die Daten für die restlichen Abschnitte vor und entscheidet, was dafür in Betrachtung gezogen wird (zum Beispiel nur Objekte in Blickrichtung der virtuellen Kamera).

2.2.3. Geometrieverarbeitung

Die *Geometrieverarbeitung* ist dafür zuständig, die Geometriedaten (*vertices* (Eckpunkte)) für die Rasterung vorzubereiten. Dafür müssen die *vertices* meist zuerst von ihrer lokalen Position (innerhalb des Objektes) in die Position innerhalb des Sichtfeldes der virtuellen Kamera umgewandelt werden.

Dies passiert normalerweise mithilfe mehrerer Matrizen (siehe Abbildung 2.3). Dabei wandelt die *Model*-Matrix die lokalen Vertexdaten in das globale (*world*) Koordinatensystem um. Die *View*-Matrix wandelt dann die transformierten Daten wiederum in das lokale System der virtuellen Kamera um. Schlussendlich projiziert die *Projection*-Matrix die Vertexdaten aus dem dreidimensionalen Raum auf die Bildebene.

Die *Model*-Matrix ist dabei abhängig von der Transformation des jeweiligen Objektes im *world space*. Aus der Transformation der Kamera ergibt sich die *View*-Matrix und die *Projection*-Matrix wird aus den speziellen Eigenschaften der Kamera (Sichtfeld und *near/far-plane* bei einer perspektivischen Projektion) gebildet. Manchmal werden die einzelnen Matrizen *Model*, *View* und *Projection* auch zur sogenannten *ModelViewProjection*-Matrix zusammengefasst um die Anzahl der Vektor-Matrix-Multiplikationen zu verringern.

Der Abschnitt Geometrieverarbeitung ist bis auf die finale Umwandlung vom *clip space* in den *screen space* frei programmierbar. Dabei wird ein zugewiesener Vertex-*Shader* für jeden Vertex aufgerufen. Da dieser Abschnitt aber auf der GPU läuft, kann man die Geometrieverarbeitung aber nicht mit einer üblichen Programmiersprache für CPUs implementieren, sondern muss eine Programmiersprache für *Shader*, wie

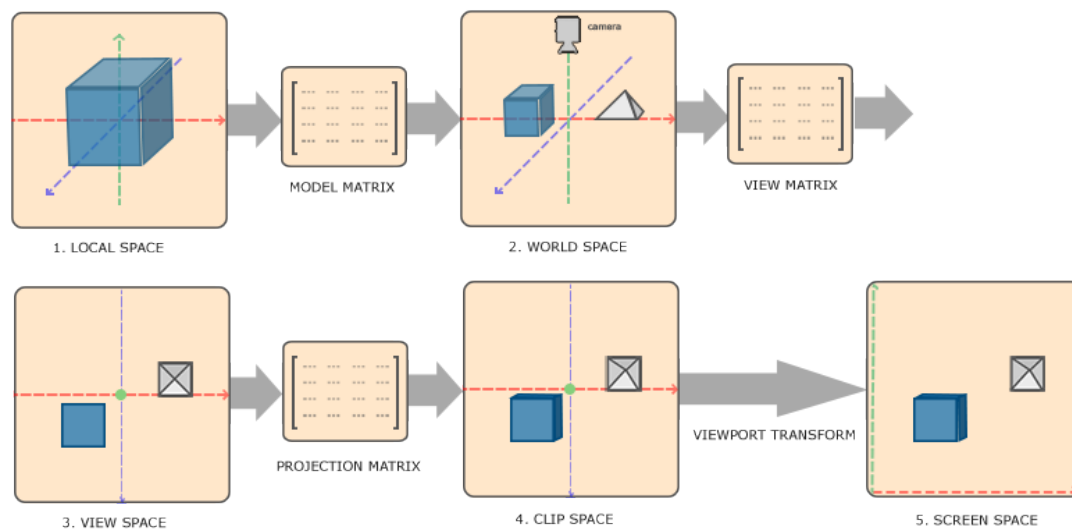


Abbildung 2.3.: Transformation von Objekt bezogenen Vertexdaten in den *world space*, dann in den *view space* und schlussendlich in den *clip space*. Die finale Umwandlung vom *clip space* in den *screen space* passiert dabei automatisch und verschiebt den Ursprung von der Mitte in entweder die ober oder untere (abhängig von der verwendeten Grafik-API) linke Ecke. Quelle: https://learnopengl.com/img/getting-started/coordinate_systems.png [22]

zum Beispiel GLSL verwenden. Die Syntax von GLSL ist dabei stark an der von C angelehnt.

In Listing 2.1 sieht man die Implementierung der Geometrieverarbeitung in der, bei dieser Arbeit entstandenen, **spider**-Engine. Dabei werden die Matrizen *Model*, *View* und *Projection* einzeln dem Shader übergeben und sind für alle Shader-Ausführungen (für dieses Objekt) konstant. Jedoch unterscheiden sich bei jeder Ausführung die Daten des jeweiligen Vertex, für den der *Shader* ausgeführt wird. Diese spezifischen Daten bekommt der Shader in den 4 Eingangsparametern ab Zeile 14 übergeben:

- *inPosition*: die Position des Vertex innerhalb des Objektes
- *inTexCoords*: die Texturkoordinaten des Vertex, zur späteren Bestimmung von Materialeigenschaften in der Pixelverarbeitung
- *inNormal*: die Normale des Vertex, meist aus den Normalen der angrenzenden Polygonflächen berechnet
- *inTangent*: die Tangente des Vertex, wird zur Anwendung von *normal maps* in der Pixelverarbeitung benötigt

Die 4 Ausgangsparameter ab Zeile 19 werden dann später, nach der Interpolation bei der Rasterung, in der Pixelverarbeitung benutzt. Wobei der Prefix *frag* hier für *fragment* steht, was eine alternative Bezeichnung für einen Pixel ist.

- *fragPosWorld*: die Position des Pixels im *world space*
- *fragTexCoords*: die Texturkoordinaten des Pixels
- *fragNormal*: die Normale des Pixels
- *fragTangent*: die Tangente des Pixels

gl_Position ist ein von GLSL reservierter Ausgangsparameter in dem die berechnete Vertexposition, zur weiteren Nutzung in folgenden Pipeline-Abschnitten, im *clip space* gespeichert werden soll. Die lokale Vertexposition wird dabei zuerst vom lokalen System des Objektes mithilfe der *Model*-Matrix in den *world space* transformiert. Diese Position wird dann einerseits direkt in *fragPosWorld* abgespeichert und andererseits mithilfe der *View*- und *Projection*-Matrizen in den *clip space* transformiert und in *gl_Position* gespeichert. Der Wert für *fragTexCoords* wird direkt von *inTexCoords* übernommen, da er unabhängig vom jeweiligen Koordinatensystem ist. Da *inNormal* und *inTangent* jeweils normalisierte Vektoren (also Richtungen) darstellen, können diese nicht mit der normalen *Model*-Matrix in den *world space* transformiert werden. Dazu muss erst die Transponierte der inversen *Model*-Matrix gebildet werden. Mit dieser können dann *fragNormal* und *fragTangent* jeweils von *inNormal* und *inTangent* berechnet werden.


```

1 #version 450
2 #extension GL_ARB_separate_shader_objects : enable
3
4 layout(set = 0, binding = 0) uniform Model {
5     mat4 model;
6 };
7
8 layout(set = 0, binding = 1) uniform Camera {
9     mat4 view;
10    mat4 proj;
11    vec3 pos;
12 } cam;
13
14 layout(location = 0) in vec3 inPosition;
15 layout(location = 1) in vec2 inTexCoords;
16 layout(location = 2) in vec3 inNormal;
17 layout(location = 3) in vec3 inTangent;
18
19 layout(location = 0) out vec3 fragPosWorld;
20 layout(location = 1) out vec2 fragTexCoords;
21 layout(location = 2) out vec3 fragNormal;
22 layout(location = 3) out vec3 fragTangent;
23
24 void main() {
25     vec3 pos_world = vec3(model * vec4(inPosition, 1.0));
26     gl_Position = cam.proj * cam.view * vec4(pos_world, 1.0);
27     fragPosWorld = pos_world;
28     fragTexCoords = inTexCoords;
29     mat3 to_world = mat3(transpose(inverse(model)));
30     fragNormal = to_world * inNormal;
31     fragTangent = to_world * inTangent;
32 }

```

Listing 2.1: Beispiel eines Vertex Shaders in GLSL, welcher so in der **spider**-Engine verwendet wird

2.2.4. Rasterung

Nach der *Geometrieverarbeitung* sind nun alle Vertices des Polygons (im Folgenden zur Vereinfachung als Beispiel ein Dreieck) in den *screen space* transformiert worden. Falls die Vertices im Bild liegen, d.h. sie haben sowohl im vertikalen, als auch horizontalen Bereich des *screen space* Werte zwischen 0 und 1, müssen nun die Pixel bestimmt werden, die in dem Dreieck liegen. Bei Polygonen die teilweise innerhalb und teilweise außerhalb des sichtbaren Bereichs liegen, wird das Polygon automatisch so zerstückelt, dass nur die sichtbaren Teile berücksichtigt werden (siehe Abbildung 2.4).

Die *Rasterung* ist hierbei fest in der Hardware einer GPU verbaut und kann nur per Parameter konfiguriert werden. Dazu zählen zum Beispiel:

- Art der zu rasternden Primitive: Punkte, Linien oder Dreiecke

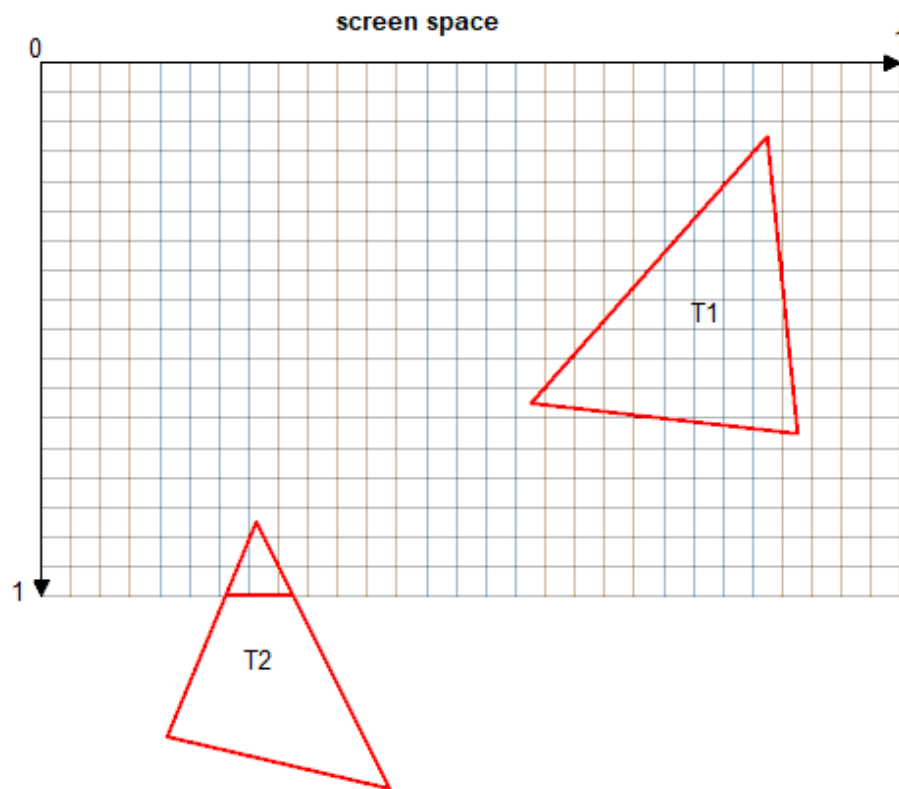


Abbildung 2.4.: Zwei Dreiecke **T1** und **T2** im *screen space*. **T1** wird hierbei komplett gerastert und **T2** erst so zerteilt, dass nur der sichtbare Bereich gerastert werden muss.

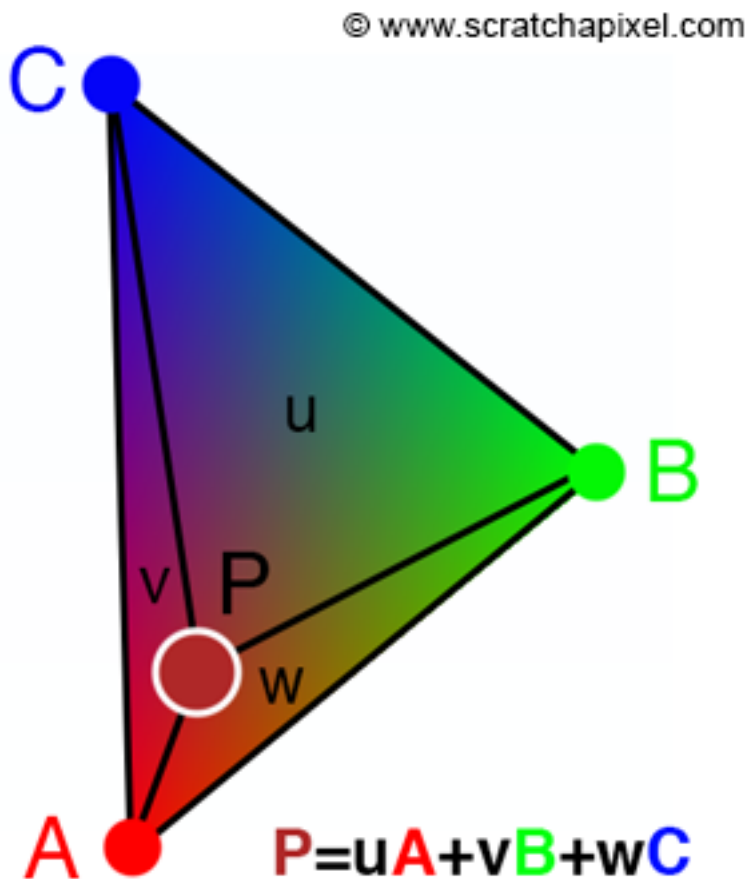


Abbildung 2.5.: Interpolation von Vertexdaten innerhalb eines Dreiecks am Beispiel der Farbinformation der Vertices mithilfe von baryzentrischen Koordinaten [35]

- Wie soll die Vorderseite eines Dreiecks bestimmt werden? Vertices sind entweder im oder gegen den Uhrzeigersinn angeordnet
- Welche Seiten eines Dreiecks sollen berücksichtigt werden? Nur die Vorderseite, nur die Rückseite oder beide?

Nun wird für jeden Pixel, der im Polygon liegt, die Vertexeigenschaften, d.h. die Ausgangsparameter der Geometrieverarbeitung, zwischen den Vertices des Polygons interpoliert (siehe Abbildung 2.6). Für jeden dieser Pixel wird dann mit den interpolierten Eigenschaften als Eingangsparameter der *Pixel-Shader*, oder auch *Fragment-Shader*, zum finalen Abschnitt der Pixelverarbeitung aufgerufen.

2.2.5. Pixelverarbeitung

In der *Pixelverarbeitung* wird der finale Farbwert eines Pixels berechnet im *Framebuffer* gespeichert. Der *Framebuffer* besteht dabei aus einem (oder auch mehreren) *color buffer*, was im Prinzip ein zwei dimensionales Array (in Größe der Pixelanzahl: Breite * Höhe) für Farbwerte ist. Dazu kommt ein gleich großer *depth buffer* der für jeden Pixel einen Tiefenwert (Distanz zur Kamera) abspeichert. Mit dem *depth buffer* kann somit pro Pixel verglichen werden, ob der aktuell zu bearbeitende Punkt näher an der Kamera (Farbwert des Punktes wird berechnet und überschreibt den Tiefenwert) oder weiter weg ist (Punkt wird nicht weiter beachtet).

Der *depth buffer* bietet hierbei den Vorteil, dass die Polygone nicht von „hinten (fern) nach vorne (nah)“ gezeichnet werden müssen. Dadurch, dass jeder Punkt nur mit dem aktuell der Kamera am nächsten verglichen werden muss, können die Polygone in beliebiger Reihenfolge bearbeitet werden. Dies gilt allerdings nur für opake, nicht transparente, Objekte. Bei teil-transparenten Objekte muss trotzdem noch auf die Zeichenreihenfolge geachtet werden.

Falls der gerasterte Pixel erfolgreich mit dem aktuellen Tiefenwert verglichen wurde, d.h. er ist näher an der Kamera wird für ihn der *Pixel-Shader*, im folgenden als *Fragment-Shader* bezeichnet, aufgerufen. Als Beispiel kann hier der *Fragment-Shader* für PBR in der **spider**-Engine gesehen werden (siehe Anhang B). Der *Fragment-Shader* berechnet dabei aus den interpolierten Vertexdaten und anderen Eingaben (zum Beispiel zusätzliche Daten wie Texturen) einen Farb- und Transparenzwert. Auch kann hier der Pixel noch verworfen werden, wenn zum Beispiel ein Transparenzwert von 0 (voll-transparent) berechnet wurde.

Die Pixelverarbeitung kann man also wiederum in drei Teilabschnitte unterteilen:

- 1) Vergleichen mit dem aktuellen Tiefenwert aus dem *depth buffer*
- 2) Berechnen des Farbwertes
- 3) Zusammenfügen mit dem aktuell im *color buffer* gespeicherten Farbwert

Dabei ist hier nur der Teilabschnitt 2) frei per *Shader* programmierbar. Teilabschnitte 1) und 3) können ähnlich wie im Abschnitt Rasterung (2.2.4) mit Parametern konfiguriert werden.

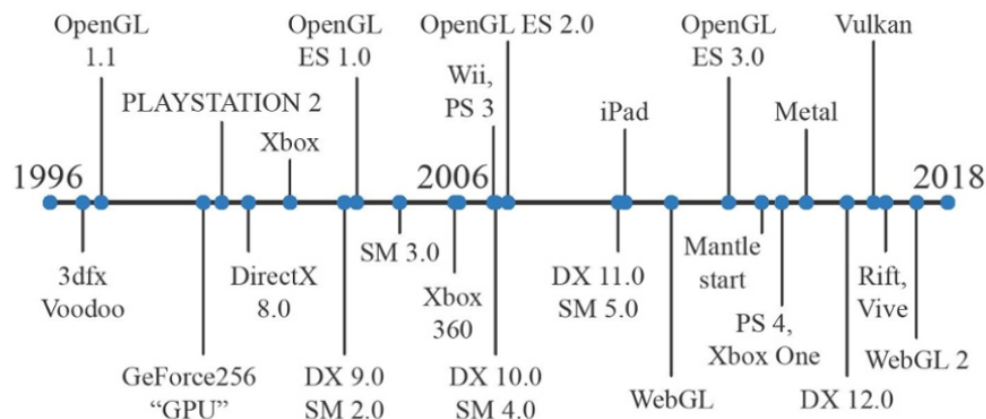


Abbildung 2.6.: Zeitlinie mit wichtigen Grafik-API- und Hardwareerscheinungen [2, S. 38]

2.3. Grafik-APIs (Vgl. 2, The Evolution of Programmable Shading and APIs)

Der Begriff „GPU“ wurde erstmals 1999 von NVIDIA eingeführt, jedoch werden im folgenden der Einfachheit halber alle vorherigen Chips mit ähnlicher Funktion unter dem Begriff „GPU“ zusammengefasst.

Analog zu den Programmiersprachen für CPUs, wurden auch Ideen zur Programmierung von GPUs, entwickelt bevor es die Hardware unterstützt hat. So wurden frei programmierbare *Shader* wurde erstmals in *Cooks Shade Trees* [3] erwähnt. Daraufhin wurde in den späten 1980er Jahren die *RenderMan Shading Language* [34] entwickelt, welche auch heute noch in der Filmproduktion eingesetzt wird. Zu diesem Zeitpunkt waren GPUs aber nichts weiter wie fest in die Hardware implementierte Bildspeicher mit einfachsten Funktionen [Vgl. 24].

Die erste GPU zur Beschleunigung von 3D-Szenen, die 3dfx Voodoo [41], hatte die Rendering-Pipeline direkt in die Hardware implementiert. Auch die NVIDIA GeForce 256 [42], die erste GPU, die auch tatsächlich so bezeichnet wurde, konnte nicht programmiert werden. Dafür konnte die integrierte Rendering-Pipeline zumindest konfiguriert werden.

Im Jahre 2001 ist mit der NVIDIA Geforce 3 [43] die erste teilweise programmierbare GPU erschienen. Durch die Grafik-APIs Microsoft DirectX 8.0 und das nicht-proprietäre OpenGL konnte hier die *Vertex-Shader* mit einer Assembly-artigen Sprache programmiert werden. Mit DirectX 8.0 wurde auch das Konzept des *Shader Model (SM)* [Vgl. 44, Shader model comparison] eingeführt, welches spezifiziert

welche Eigenschaften (zum Beispiel, wie viele Register dem Shader-Programm zur Verfügung stehen), die jeweilige Hardware erfüllen muss. Zwar unterstützte DirectX 8.0 auch programmierbare *Pixel-Shader* (in OpenGL *Fragment-Shader* genannt), jedoch waren diese noch sehr eingeschränkt im Funktionsumfang.

Dabei funktionieren GPUs bis heute so, dass das gleiche Shader-Programm gleichzeitig auf mehrere Eingabedaten mit der selben Struktur, zum Beispiel Vertexdaten bei einem *Vertex-Shader*, angewandt wird. Dadurch konnten in den Shader-Programmen für die ersten GPUs noch keine Verzweigungen benutzt werden, da diese ja potentiell zu unterschiedliche Anweisungen für die jeweiligen Daten führen. Dies wurde von Shader-Programmierern dadurch umgangen, dass einfach alle Anweisungen aller potentiellen Verzweigungen nacheinander ausgeführt wurden und danach die nicht gebrauchten Ergebnisse verworfen wurden. Das führte zwar zu „nutzlosen“ Berechnungen, jedoch war dies durch die Parallelität insgesamt immer noch schneller, als die Daten sequentiell zu verarbeiten.

Mit dem Erscheinen von DirectX 9.0 mit Shader Model 2.0 im Jahre 2002, hatten Programmierer endlich die volle Kontrolle über sowohl *Vertex-* als auch *Fragment-Shader*. Für den offenen Standard OpenGL wurde diese Möglichkeit relativ zeitgleich durch *extensions* ermöglicht. Außerdem konnten die Shader nun in einer C ähnlichen Sprache geschrieben werden: High Level Shading Language (HLSL) [44] für DirectX und GLSL [44] für OpenGL. Mit der Einführung des Shader Model 3.0 im Jahre 2004 konnte nun endlich auch ein dynamischer Programmablauf mit Verzweigungen im Shader benutzt werden. Mit der Einführung der Spielekonsole Nintendo Wii Ende 2006 wurde auch eine der letzten GPUs mit fester Rendering-Pipeline eingeführt.

Ebenfalls wurde Ende 2006 DirectX 10.0 und Shader Model 4.0 eingeführt, welche eine neue Art von Shadern (*Geometry-Shader*) und die Vereinheitlichung des Shader-Designs mit sich brachte. Mit DirectX 11.0 und Shader Model 5.0 im Jahre 2009 wurden schließlich unter anderem *Compute-Shader* eingeführt. Die Version von OpenGL mit ähnlichem Funktionsumfang ist dabei 4.3.

Bis dahin war das Paradigma für Grafik-APIs dem Entwickler so viel Arbeit wie möglich abzunehmen und zum Beispiel im Treiber Eingaben zu validieren und Ressourcen (zum Beispiel Texturen und Buffer) automatisch zu verwalten. Dies änderte sich 2013 mit der Einführung von Mantle durch den CPU- und GPU- Hersteller Advanced Micro Devices, Inc (AMD) als Alternative zu Direct3D und OpenGL. Die Idee hinter der API, die in Zusammenarbeit mit dem Videospielentwickler DICE entwickelt wurde, war es, den Overhead im Grafiktreiber so viel wie möglich zu reduzieren und diese Kontrolle dem Entwickler zurückzugeben. Microsoft hat die Ideen hinter Mantle aufgenommen und 2015 mit DirectX 12.0 eine DirectX Version dieses neuen Paradigmas veröffentlicht. Wichtig ist hier, dass DirectX 12 keinen Fokus auf neuen Funktionen hat, sondern nur die Verwendung der API radikal geändert hat. Bei der Einführung von DirectX 12.0 hatte es die gleichen Funktionalitäten.

litäten wie das damals aktuelle DirectX 11.3. Auch Apple hat 2014 mit Metal ihre eigene „low-overhead“ API veröffentlicht. Diese debütierte zuerst auf den mobilen Geräten wie iPhone und iPad, war jedoch ein Jahr später auch auf Macs verfügbar. Währenddessen hatte AMD ihre Arbeit zu Mantle an das für OpenGL zuständige Konsortium Khronos Group gestiftet. Diese führte 2016 mit Vulkan ihre Version einer „low-overhead“ API ein. Ähnlich wie bei den DirectX-Versionen 11 und 12, hat Vulkan hier OpenGL nicht abgelöst, sondern stellt nur eine Alternative dar. Mit Vulkan wurde auch eine Zwischensprache (engl. *intermediate language*) für Shader-Programme, ähnlich wie LLVM IR, mit dem Namen SPIR-V eingeführt, die es dem Entwickler erlaubt Shader vor zu kompilieren.

Für mobile Geräte gibt es OpenGL ES („Embedded Systems“) welches 2003 als eine im Funktionsumfang verringerte Version von OpenGL 1.3 veröffentlicht wurde. OpenGL ES 1.0 beschrieb noch eine feste Rendering-Pipeline, welche aber durch Version 2.0 im Jahr 2007 die Möglichkeiten zur Shader-Programmierung erhielt. OpenGL ES 3.0 wurde schließlich 2012 eingeführt und brachte viele bekannte Funktionen der „großen Brüder“ OpenGL und DirectX mit.

Die Grafik-API für Webbrowser WebGL basiert wiederum auf OpenGL ES, wurde 2011 veröffentlicht und ist mit Version 2.0 von OpenGL ES im Funktionsumfang äquivalent. WebGL 2.0 wurde 2017 eingeführt und nutzt OpenGL ES 3.0.

3. WebGPU

3.1. Entstehung (Vgl. 46)

Da WebGL auf dem alten API-Paradigma von OpenGL und DirectX vor Version 12 beruht, wurden Rufe laut, dass es auch eine „moderne“ Grafik-API für Webbrowser geben sollte, um deren Vorteile auch im Web nutzen zu können.

Die ersten Vorstöße dazu gab es als Google Mitarbeiter bei einer Präsentation Mitte 2016 für die WebGL Arbeitsgruppe über die grundlegenden Ideen und Prinzipien für eine moderne Grafik-API fürs Web, damals noch als „WebGL Next“ bezeichnet, sprachen. Diese Ideen wurden dann Anfang 2017 von der Khronos Group aufgegriffen und in einem Meeting hat ein Google Team dann ihren mit „NXT“ betitelten Prototypen präsentiert, welcher im Webbrowser Chromium lief und sich an Konzepten von Vulkan, DirectX 12 und Metal bediente. Teams von Apple und Mozilla hatten außerdem Prototypen für Safari und Servo (die Browser Engine für Firefox) gebaut, welche sich an Apples Metal API orientiert haben.

Kurze Zeit später wurde die W3C Gruppe „GPU for the Web“ gegründet und „WebGPU“ als Arbeitstitel für die neue API beschlossen [14]. Nachdem die meisten oberflächlichen Probleme im Standardisierungsprozess Mitte 2018 gelöst waren, gab Googles Chrome Team bekannt, dass sie den zukünftigen **WebGPU** in ihren Webbrowser implementieren möchten.

3.2. Beschreibung

„The mission of the GPU on the Web Community Group is to provide an interface between the Web Platform and modern 3D graphics and computation capabilities present in native system platforms. The goal is to design a new Web API that exposes these modern technologies in a performant, powerful and safe manner. It should work with existing platform APIs such as Direct3D 12 from Microsoft, Metal from Apple, and Vulkan from the Khronos Group. This API will also expose the generic computational facilities available in today's GPUs to the Web, and investigate shader languages to produce a cross-platform solution.“

– GPU for the Web Community Group [10]

WebGPU soll hierbei ähnlich wie bei Vulkan und OpenGL, und DirectX 12 und 11 nicht WebGL direkt ablösen, sondern erstmal nur eine Alternative bilden.

3.3. WGSL

3.3.1. Übersicht

Mit WGSL [13] soll auch **WebGPU** eine eigene Shader-Sprache bekommen. Bisherige Implementierungen unterstützen jedoch bisher nur SPIR-V. Dadurch konnte der Verfasser leider keine praktischen Erfahrungen bei der Verwendung von WGSL machen. Die **spider**-Engine, die diese Arbeit begleitet hat, verwendet daher GLSL-Shader, die zu SPIR-V kompiliert werden. Trotzdem werden im Folgenden die Grundprinzipien und Beispiele zu WGSL besprochen, da es in Zukunft wohl ein wichtiger Bestandteil von **WebGPU** sein wird.

3.3.2. Kritik an WGSL

WGSL soll „trivial von und zu SPIR-V konvertierbar“ [Vgl. 13, Goals] sein, was unter anderem die Frage aufwirft, warum dann WGSL überhaupt benötigt wird [Vgl. 17]. Durch die Nähe zu der Zwischensprache SPIR-V ist die Syntax von WGSL für Entwickler, die mit einer bisherigen Shader-Sprache vertraut sind, nicht sehr üblich. Auch gibt es nicht die aus den meisten Programmiersprachen bekannten Schleifenstrukturen, sondern nur eine `loop`-Anweisung, aber nicht direkt `for`, `while` oder `do-while` [Vgl. 18]. In Zukunft soll nur noch WGSL als Shader-Sprache akzeptiert werden. Dazu müssten die bei vielen Entwicklern schon bestehenden GLSL- oder HLSL-Shader erst konvertiert werden. Dies würde in etwa so aussehen:

GLSL/HLSL -> SPIR-V -> WGSL -> *WebGPU* -> SPIR-V/DXIL/MSL

Für die Umwandlung von GLSL/HLSL zu SPIR-V gibt es bereits zahlreiche Tools, welche produktiv getestet und eingesetzt wurden, wie zum Beispiel, das in diesem Projekt genutzte *glslc*, welches in der Shader-Werkzeugsammlung *shaderc* [36] enthalten ist. Der abschließende Schritt, bei dem **WebGPU** den WGSL-Shader-Code wieder in SPIR-V/DXIL/MSL umwandeln muss, ist nötig, da WebGPU auf den jeweiligen nativen Grafik-APIs (Vulkan, DirectX 12, Metal) aufbaut, welche keinen WGSL-Shader-Code nutzen können.

Die Ursache für die Entscheidung für WGSL und gegen SPIR-V als primäres Shader-Code-Format, liegt wohl einerseits an der für das Web übliche Praxis, von Menschen lesbare Datenformate zu verwenden (kein Binärcode). Andererseits scheint es wohl rechtliche Spannungen zwischen Apple und der für SPIR-V verantwortlichen Khronos Group zu geben [Vgl. 33].

3.3.3. Datentypen

Skalare Typen

Typ	Beschreibung	GLSL	HLSL
Boolean			
bool	Entweder <code>true</code> oder <code>false</code>	bool	bool
Numerisch			
i32	32-bit Integer mit Vorzeichen	int	int
u32	32-bit Integer ohne Vorzeichen	uint	uint
f32	32-bit Fließkommazahl nach IEEE 754	float	float
-	64-bit Fließkommazahl nach IEEE 754	double	double

Tabelle 3.1.: Skalare Typen in WGSL mit den äquivalenten Typen in GLSL und HLSL

Mehrkomponenten Typen

Vektortypen

Vektortypen werden in WGSL mit

`vecN<T>`

deklariert, wobei N für die Anzahl der Komponenten und T für den Datentyp steht.

N muss dabei in $\{2, 3, 4\}$ und T muss einer der skalaren Typen aus Abschnitt 3.3.3 sein.

Das äquivalente Gegenstück hierzu in GLSL wäre

$T\text{vec}N$

wobei T hier als Abkürzung eines skalaren Typen fungiert ($b = \text{bool}$, $i = \text{int}$, $u = \text{uint}$, $d = \text{double}$). Wenn T weggelassen wird, wird der skalare Typ `float` verwendet, da dies der mit Abstand meist verwendete skalare Typ in der Shader-Programmierung ist. N muss hier ebenfalls in $\{2, 3, 4\}$ sein.

Bei HLSL werden Vektortypen wie folgt deklariert:

TN

Wobei T hier einer der skalaren Typen sein muss und N entweder in $\{1, 2, 3, 4\}$ oder gar nicht vorkommen darf. Das Weglassen von N oder $N = 1$ resultieren beide in dem skalaren Typen T .

So würde ein 3-komponentiger Vektor vom Typ `f32` bzw. `float` in den jeweiligen Sprachen so aussehen:

- WGSL: `vec3<f32>`
- GLSL: `vec3`
- HLSL: `float3`

Matrixtypen

Analog zu Vektortypen werden Matrixtypen in WGSL mit

$\text{mat}N \times M < T >$

deklariert, wobei N hier für die Anzahl der Spalten und M für die Anzahl der Zeilen steht. N und M müssen wiederum beide in $\{2, 3, 4\}$ sein. T ist ein skalarer Typ aus Abschnitt 3.3.3.

In GLSL gibt es dafür

$\text{mat}N \times M$

und als Abkürzung für eine Matrix mit quadratischer Form (Anzahl Spalten = Anzahl Zeilen)

$\text{mat}N$

Matrixkomponenten sind immer vom Typ `float` oder `double`. Eine Matrix mit doppelter Genauigkeit wird mit `dmatN×M` bzw `dmatN` deklariert.

In HLSL werden die Matrixtypen ähnlich wie die Vektortypen deklariert:

`TN×M`

wobei hier wie bei GLSL für T jeder der skalare Typen benutzt werden kann. N steht für die Anzahl der Spalten und M für die Anzahl der Zeilen und beide müssen jeweils in $\{1, 2, 3, 4\}$ sein.

Eine quadratische Matrix mit 3 Spalten und 3 Zeilen vom Typ `f32` bzw. `float` wird also jeweils so erstellt:

- WGSL: `mat3x3<f32>`
- GLSL: `mat3x3` bzw. `mat3`
- HLSL: `float3x3`

Um wiederum eine Matrix mit 3 Spalten und 2 Zeilen vom Typ `i32` bzw. `int` zu erstellen werden folgende Deklarationen benötigt:

- WGSL: `mat3x2<i32>`
- GLSL: -
- HLSL: `int3x2`

Swizzling

Um auf die Komponenten eines Mehrkomponenten Typen zuzugreifen haben diese vordefinierte Felder. So hat ein 4-komponentiger Vektor zum Beispiel die Felder `x`, `y`, `z`, `w`. Da aber 4-komponentige Vektor auch häufig für Farbwerte benutzt werden, gibt es zusätzlich noch die Felder `r`, `g`, `b`, `a`, wobei dies aber nur andere Namen für die gleichen Komponenten sind: `x = r`, `y = g`, `z = b`, `w = a`. GLSL hat zusätzlich noch das Namensset `s`, `t`, `p`, `q`, das im Zusammenhang mit Texturkoordinaten benutzt wird.

Eine Besonderheit in vielen Shader-Sprachen ist es, dass diese Felder frei kombiniert werden können um die einzelnen Komponenten umzustrukturieren, was *swizzling* [45] genannt wird (siehe Listing 3.1).

```
1 vec4 some_vec = vec4(1.0, 2.0, 3.0, 4.0);
2 // Get the second component
3 float val = some_vec.y;
4 // Get the fourth as the first, and the first as the second component of a new vec2
5 vec2 other_vec = some_vec.ar;
6 // Write to the first, third and second component
7 some_vec.xzy = vec3(0.0, some_vec.xx);
8 // But you can't mix name sets
9 vec3 faulty_vec = some_vec.xrr;
```

Listing 3.1: „Swizzling“ von Komponenten in GLSL

Dabei wird es in WGSL nicht möglich sein, in mehrere Komponenten gleichzeitig zu schreiben, also einen „swizzled“ Ausdruck auf der linken Seite einer Zuweisung zu haben [Vgl. 13, Typed Storage].

3.4. Der aktuelle Stand

3.4.1. Spezifikation

Die Spezifikation [12] der **WebGPU**-Application Programming Interface (dt. Programmierschnittstelle) (API) ist zum Zeitpunkt des Verfassens dieses Abschnittes (09.06.2020) noch nicht abgeschlossen und wurde auch während dem Verlauf der Arbeit mehrmals erweitert und angepasst.

3.4.2. Implementierung

Der aktuelle Stand der Implementierung in Webbrowsern kann unter [11] eingesehen werden.

Chromium (8) / Dawn (9)

Dawn ist eine Open-Source **WebGPU**-Implementierung, die in dem Open-Source Webbrowser Chromium verwendet wird. Unter anderem basieren die bekannten Webbrowser *Google Chrome*, *Opera* und seit neustem auch *Microsoft Edge* auf Chromium.

Die „native“ Implementierung von *Dawn* benutzt dafür die Grafik-APIs der jeweils ausführenden Plattform:

- *D3D12* auf Windows 10
- *Metal* auf macOS und iOS

- *Vulkan* auf Windows, Linux und Google eigenen Betriebssystemen (ChromeOS, Android, Fuchsia)
- *OpenGL* wo verfügbar

Das heißt, dass die **WebGPU**-Befehle hier im Hintergrund die jeweiligen Befehle der nativen Grafik-API aufrufen. Dies hat den Vorteil, dass **WebGPU** so keinen eigenen Treiber braucht, um mit der GPU zu kommunizieren. Da die generelle **WebGPU**-API-Struktur auch ähnlich zu der Struktur der nativen APIs (bis auf OpenGL) ist, bedeutet dies auch keinen großen Mehraufwand bei der Implementierung und kann sogar teilweise von Codegeneratoren bewerkstelligt werden.

Firefox Nightly (28) / wgpu (29)

wgpu ist ebenfalls eine Open-Source **WebGPU**-Implementierung in der Programmiersprache *Rust*. *wgpu* wird hierbei unter anderem im *Mozilla Firefox*-Webbrowser verwendet und von der *Mozilla Foundation* federführend entwickelt.

3.5. Nutzen außerhalb von Grafik

4. Implementierung einer WebGPU-Applikation

Für das in diesem Kapitel beschriebene Projekt wurde Windows 10 mit Ubuntu 19.10 per Windows Subsystem for Linux (WSL) [27] verwendet. Dabei wurden die Quellcode-dateien in Windows erstellt und bearbeitet, aber in Ubuntu benutzt (kompilieren, linken usw.). Deshalb beziehen sich im Folgenden alle Angaben zur Installation oder Benutzung von Werkzeugen auf die jeweilige Linux-Version.

4.1. spider

*In diesem Abschnitt wird mit „der Benutzer“, ein Benutzer der **spider**-Engine verstanden, also eine Person, die mithilfe der Engine eine eigene Applikation entwickelt. „Der Benutzer“ sollte hier als neutrale Form, die die weibliche, sowie die männliche Form beinhaltet, verstanden werden.*

4.1.1. Überblick

Mit der parallel zu dieser Arbeit entstandenen **spider**-Engine, lässt sich mit geringem Aufwand eine Webapplikation zur Darstellung von 3D-Szenen in C/C++ erstellen. Das Grundgerüst dafür ist sehr minimal (siehe Listing 4.1).

```

1 #include "spider/spider.h"
2
3 // Create the initial state of your scene
4 void initApplication();
5 // Update your scene each frame
6 bool update(float delta_time_s);
7
8 int main() {
9     const uint32_t surface_width = 1280;
10    const uint32_t surface_height = 720;
11
12    // Initialize the spider engine
13    spInit(&(SPInit){
14        .surface_size = {
15            .width = surface_width,
16            .height = surface_height
17        },
18        .update_func = update,
19        .camera = {
20            .pos = {0.0f, 2.0f, 5.0f},
21            .look_at = {0.0f, 0.0f, 0.0f},
22            .mode = SPCameraMode_LookAt,
23            .fovy = glm_rad(60.0f),
24            .aspect = (float)surface_width / (float)surface_height,
25            .near = 0.1f,
26        },
27    });
28
29    initApplication();
30    spStart();
31
32    return 0;
33 }
34
35 void initApplication() {
36     // Here you can create lights (only spotlights for now),
37     // create custom meshes and materials
38     // or load them from glTF files (recommended)
39 }
40
41 bool update(float delta_time_s) {
42     // Here you can update your created lights, objects and the main camera
43
44     // Return false, if you want to quit the application
45     return true;
46 }

```

Listing 4.1: Grundgerüst einer Applikation mit der **spider**-Engine

Die **spider**-Engine ist darauf ausgelegt, mit möglichst wenig Code auf Seiten des Benutzers eine 3D-Applikation zu erstellen. So lässt sich in rund 150 Zeilen Code eine interaktive 3D-Szene mit einem UserInterface (dank integriertem Dear ImGui [4]) erstellen (siehe Abbildung 4.1 und Anhang A). Trotzdem ist die Funktionalität der **spider**-Engine sehr beschränkt, wenn man etwas anderes als das Darstellen von 3D-Modellen und eines einfachen UserInterfaces will. Jedoch ist der Verfasser überzeugt, dass es für einfache Prototypen ausreichend ist.

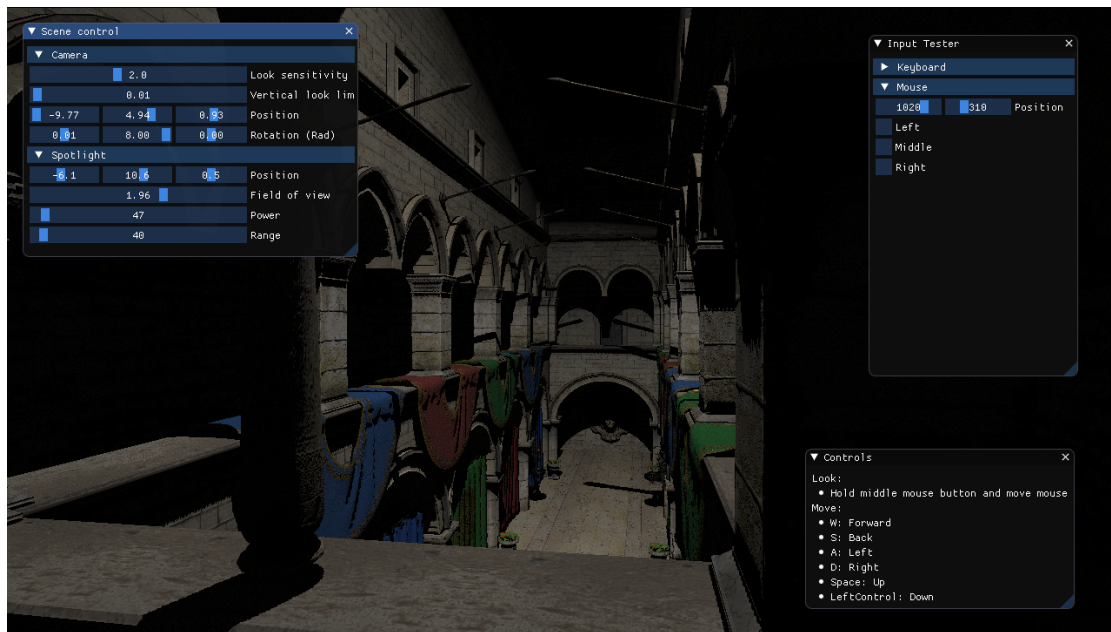


Abbildung 4.1.: Interaktive 3D-Szene mit UserInterface. Bildschirmaufnahme durch Verfasser.

4.1.2. Codestyle

Für das Projekt wurde der C Standard C99 und der von **Emscripten** verwendete Compiler clang benutzt. Folgende Punkte sollten in allen größeren C-Compilern, die C99 unterstützen, problemlos funktionieren, wurden dort aber nicht getestet.

Um einen einheitlichen Codestyle zu erhalten, hat der Verfasser sich auf folgende Punkte festgelegt:

stdint.h* und *stdbool.h

Im Projekt wurden die seit C99 verfügbaren Headerdateien `stdint.h` und `stdbool.h` benutzt, um einerseits mehr Kontrolle über den Speicherbedarf der jeweiligen Komponenten zu erlangen und andererseits mehr Kontext zu geben. So ist zum Beispiel bei einer Variable des Typen `uint8_t` schnell klar, dass der Verfasser für die Benutzung der Variable nur einen relativ kleinen Bereich an positiven Ganzzahlen beabsichtigt hat. Auch der Typ `bool` ist im Vergleich zu einem `int` eindeutig im Bezug auf die Absicht des Verfassers. Interessanterweise wurde beim gesamten Projekt (außer bei der Kommunikation mit den verwendeten Bibliotheken) nur Ganzzahldatentypen ohne Vorzeichen benutzt (`uint8_t`, `uint16_t`, `uint32_t` und `uint64_t`).

Typ	Prefix	Beispiel
„öffentlich“		
Struktur	SP	SPMesh
Enum	SP	SPKey
Funktion	sp	spInit
Preprozessor directives	SP_	SP_INVALID_ID
globale Variable	sp_	<i>kein Beispiel</i>
„privat“		
Struktur	_SP	_SPRenderPipeline
Enum	_SP	<i>kein Beispiel</i>
Funktion	_sp	_spSetupPools
Preprozessor directives	_SP_	_SP_MATERIAL_POOL_DEFAULT
globale Variable	_sp_	_sp_state

Tabelle 4.1.: Verwendete Prefixe

Prefixe

Da es in C keine Namensbereiche, wie zum Beispiel in C++, gibt, werden alle Strukturen, Enums, Funktionen, Preprozessor directives und globale Variablen mit einem Prefix versehen (siehe Tabelle 4.1). Die Unterscheidung zwischen *öffentlich* und *privat* ist hierbei keine Unterscheidung auf Compilerlevel, sondern nur für den Benutzer, damit dieser erkennen kann, welche Funktionen und Strukturen benutzt werden sollten und welche nur intern genutzt werden.

Definition von Strukturen

Strukturen werden im Projekt wie in Listing 4.2 definiert.

```

1 typedef struct MyStructure {
2     uint32_t x;
3     struct {
4         float width;
5         float height;
6     } size;
7 } MyStructure;
```

Listing 4.2: Definition von Strukturen

Dies hat den Vorteil, dass die so definierten Strukturen nicht immer ein vorhergehendes `struct` bei der Verwendung im Code benötigen. Zusätzlich wird direkt nach dem `struct` auch der Name der Struktur benötigt, damit diese weiterhin vorwärts deklariert werden kann [vgl. 39].

Semantisch zusammengehörige Attribute innerhalb einer Struktur werden entweder in eine extra Struktur ausgelagert, oder als anonyme Struktur (siehe `size` im Beispiel) innerhalb der Struktur definiert. Dadurch kann über `my_structure.size.width` auf das Attribut zugegriffen werden.

Desc-Argument

In vielen Programmiersprachen ist es möglich, beim Aufrufen einer Funktion nur einen Teil der Argumente anzugeben, während die restlichen Argumente mit Standardwerten initialisiert werden. C unterstützt zwar keine Standardargumente, dafür aber ab C99 *Bestimmte Initialisierer* (engl. *designated initializers*), was bedeutet, dass man die Felder von Strukturen (und Arrays) mit ihrem jeweiligen Namen (oder Index) in beliebiger Reihenfolge innerhalb einer Initialisierungsanweisung angeben kann. Wenn mindestens ein Feld initialisiert wird, werden alle nicht initialisierten Felder mit `0` initialisiert. (Zur Verdeutlichung siehe Listing 4.3).

```

1 // No guarantee on initialization values
2 MyStructure my_structure;
3
4 // Initialization in field order
5 // -> (x = 5, size.width = 3.0f, size.height = 4.0f)
6 MyStructure my_structure_order = {5, {3.0f, 4.0f}};
7 MyStructure my_structure_order2 = {5, 3.0f, 4.0f};
8
9 // Initialization in field order, rest is zero initialized
10 // -> (x = 5, size.width = 0.0f, size.height = 0.0f)
11 MyStructure my_structure_order_zero = {5};
12
13 // Initialization with field names
14 // -> (x = 5, size.width = 3.0f, size.height = 4.0f)
15 MyStructure my_structure_named = {.size.width = 3.0f, .x = 5, .size.height = 4.0f};
16 MyStructure my_structure_named2 = {.size = {.width = 3.0f, .height = 4.0f }, .x =
    5};
17
18 // Initialization with field names, rest is zero initialized
19 // -> (x = 0, size.width = 3.0f, size.height = 0.0f)
20 MyStructure my_structure_named_zero = {.size.width = 3.0f};
21 MyStructure my_structure_named_zero2 = {.size = {.width = 3.0f}};
22
23 // Initialization of array fields
24 // -> (0, 3, 0, 0, 1)
25 uint32_t array[5] = {
26     [1] = 3,
27     [4] = 1
28 };

```

Listing 4.3: Beispiele zu bestimmten Initialisierern (Unter Verwendung der in Listing 4.2 definierten Struktur *MyStructure*)

Im Projekt wird das bei Funktionen mit einer nicht trivialen Argumentliste verwendet. Dabei wird für die jeweilige Funktion eine Struktur mit dem Postfix `Desc` (für

engl. *descriptor*, analog zu den **WebGPU-Descriptor**-Strukturen) definiert. Diese enthält alle benötigten Argumente (mit möglichst sinnvollem Standardwert 0). Da es außerdem in C99 möglich ist, einen Zeiger auf ein temporäres Objekt zu erzeugen, hat die jeweilige Funktion nun als einziges Argument einen Zeiger auf ein konstantes Objekt der Desc-Struktur: `void myFunction(const MyFunctionDesc* desc);`. Nun kann die Desc-Struktur entweder vor dem Aufrufen der Funktion erschaffen und befüllt werden, oder direkt beim Funktionsaufruf initialisiert werden (siehe Listing 4.4).

```

1 typedef struct MyFunctionDesc {
2     uint32_t values[5];
3     const char* name;
4 } MyFunctionDesc;
5
6 void myFunction(const MyFunctionDesc* desc);
7
8 MyFunctionDesc my_desc;
9 my_desc.values[2] = 5;
10 my_desc.name = "Abc";
11 myFunction(&my_desc);
12
13 myFunction(&(MyFunctionDesc){
14     .values = {
15         [2] = 5,
16     },
17     .name = "Abc",
18 });

```

Listing 4.4: Verwendung einer Desc-Struktur zum Übergeben von Argumenten an eine Funktion

handles (Vgl. 38)

Die **spider**-Engine hat das Grundprinzip, dass der Benutzer sich weder um die (De-)Allokation von Speicher, noch über die Referenzierungen zwischen Objekten, kümmern muss und somit den vollen Fokus auf die Erstellung der jeweiligen Applikation richten kann. Dabei soll auch die Speicherverwaltung von erschaffenen Objekten (zum Beispiel ein `SPMesh`) einzig der **spider**-Engine überlassen werden. Dabei soll die Engine auch die Möglichkeit haben, die jeweiligen Objekte intern zu verschieben und zu sortieren (bisher nicht benutzt) um den internen Zugriff zu optimieren. Daher kann die Engine nicht direkt einen Zeiger auf ein erschaffenes Objekt liefern, da dieser bei einer Änderung der internen Struktur ungültig wird.

Dafür gibt es für jede öffentliche Struktur eine *handle*-Struktur mit Postfix ID (zum Beispiel für `SPMesh` -> `SPMeshID`) welche eine interne ID (`uint32_t id`) beinhaltet. Bisher wird die interne ID des *handle* einfach direkt als Index in das interne Array zur Verwaltung der jeweiligen Objekte benutzt, jedoch könnte man hier noch ähnlich zu [38] die unbenutzten Bits (bei zum Beispiel maximal $2^{16} = 65536$ gleichzeitig

existierenden Objekten bleiben hier 16 von 32 Bits übrig) zur Versionierung des *handle* verwenden. Um das referenzierte Objekt dann im Anwendungscode zu benutzen kann ein temporärer Zeiger erzeugt werden (siehe Listing 4.5). Dieser Zeiger sollte vom Benutzer nicht gespeichert werden und nur erzeugt werden, wenn das referenzierte Objekt wirklich verwendet (Attribute lesen oder schreiben) wird. Es gibt keine Garantie, dass der Zeiger im nächsten Update noch auf das gleiche Objekt zeigt, oder überhaupt gültig ist.

```

1 typedef struct SPObject {
2     float size;
3 } SPObject;
4
5 typedef struct SPObjectID {
6     uint32_t id;
7 } SPObjectID;
8
9 // Store the object handle
10 SPObjectID object_id;
11
12 void init() {
13     // Create object and store the handle
14     object_id = spCreateObject();
15 }
16
17 void update() {
18     // Later use the referenced object
19     SPObject* object = spGetObject(object_id);
20     // If handle is not valid, spGetObject returns a NULL pointer
21     if(object) {
22         object->size *= 2.0f;
23     }
24 }
```

Listing 4.5: Verwendung von *handles*

4.2. emscripten (5)

Augenscheinlich wurde das Projekt, das diese Arbeit begleitet hat, in C geschrieben. Da die entstandene Applikation jedoch später in einem Webbrowser laufen und dort die **WebGPU**-API benutzen soll, **<TODO>**. Dazu wird das Werkzeug **Emscripten** verwendet. Mit **Emscripten** lässt sich (unter anderem) C-Quellcode mit Hilfe von LLVM [23] zu JavaScript und WebAssembly [37] kompilieren.

4.2.1. Funktionsweise

Um **Emscripten** benutzen zu können, muss als erstes das Emscripten SDK (emsdk) heruntergeladen und installiert werden [6]. Am einfachsten ist es hierbei, das SDK im gleichen Ordner zu platzieren, wie das Projekt, das es benutzen soll:

```
path\  
└─ to\  
    └─ emsdk\  
        └─ my_project\  
            └─ test.c
```

Zum Beginn einer Terminal-Sitzung müssen noch die Umgebungsvariablen für **Em-
scripten** gesetzt werden (dies macht man am besten im `path\to\emsdk\`-Ordner):

```
source ./emsdk_env.sh
```

Im Projektordner (`path\to\my_project\`) kann nun die C-Quelldatei kompiliert werden:

```
./emcc test.c -WASM=1 -o test.html
```

4.3. Benutzte WebGPU-Features

Die benutzten Features der **WebGPU!** (**WebGPU!**)-API wurden nach mehreren Punkten ausgesucht:

- Die Implementierung sollte sowohl in Chromium Canary als auch in Firefox Nightly verfügbar sein, um vergleich- und testbar zu sein
- Es sollte einen konkreten Nutzen in der **spider**-Engine erfüllen, um das Projekt nicht unnötig kompliziert zu machen

4.4. Besonderheiten bei der Entwicklung einer GPU Applikation

Dadurch, dass viele Befehle nicht auf der CPU, sondern auf der GPU ablaufen, sind traditionelle *Debugger*, wie *gdb*, nur begrenzt nützlich zum Finden von Fehlern im Programm. Damit kann man zwar immer noch den generellen Ablauf des Programms überprüfen, aber was genau nach dem Aufrufen einer API-Funktion auf der GPU geschieht, ist damit nicht einsehbar. Dafür gibt es verschiedene Grafik-Debugger, die praktisch alle nach dem gleichen Prinzip ablaufen. So muss meist das zu testende Programm über den Grafik-Debugger gestartet werden, damit dieser sich in das Programm einklinken kann. Im Gegensatz zu klassischen Debuggern muss das Programm dabei nicht als **Debug**-Version gebaut werden. Wenn das zu testende Programm nun erfolgreich gestartet wurde, hat man die Möglichkeit einen oder mehrere *frames*(Einzelbilder) zu erfassen. Die jeweiligen *frames* werden dann

aufbereitet und man kann sich alle vom Programm getätigten API-Aufrufe und alle GPU-Ressourcen zum jeweiligen Zeitpunkt anschauen.

Durch die Besonderheit der Applikation im Bezug auf die Neuheit der **WebGPU**-API und der Tatsache, dass die Applikation im Webbrowser läuft, konnten allerdings mehrere der Grafik-Debugger nicht sinnvoll verwendet werden. So hatten die oft benutzten Programme *RenderDoc* [19] und *NVIDIA NSight Graphics* [32] Probleme damit, mit den benutzten Webbrowser-Versionen *Chrome Canary* [7] und *Firefox Nightly* [28] eine Verbindung aufzubauen. *RenderDoc* konnte dabei nur die *Direct3D 11* Befehle zur Anzeige des finalen Bildes, aber nicht die eigentlichen *Direct3D 12* Befehle zur Erstellung des Bildes, erfassen. Bei *NVIDIA NSight* war ein Erfassen von *frames* gar nicht möglich, da *D3D11on12* [25] (eine Softwareschicht um *Direct3D 11*-Befehle auf *Direct3D 12*-Befehlen zu übersetzen), das in *Chrome* benutzt wird, gar nicht unterstützt wird. Einzig mit *PIX on Windows* [26] war es möglich alle API-Aufrufe und GPU-Ressourcen richtig anzuzeigen. Jedoch auch nicht als **WebGPU** Aufrufe, sondern als *Direct3D 12* Aufrufe und nur wenn *Chrome Canary* als Webbrowser benutzt wird. Da **WebGPU** aber, wie schon angesprochen, sehr ähnlich zu *Direct3D 12* ist, ist dies kein großes Problem.

Zum korrekten Starten des *Chrome Canary* Webbrowser durch *PIX on Windows*, müssen noch zusätzliche Kommandozeilenparameter übergeben werden: `-no-sandbox -disable-gpu-sandbox -disable-gpu-watchdog -disable-direct-composition` und als API *D3D12 (ignore D3D11)* ausgewählt werden (wie in Abbildung 4.2 zu sehen). Mit **Launch** wird dann der richtig konfigurierte *Chrome Canary* gestartet und man kann nun auf die zu testende Webseite (bei diesem Projekt zum Beispiel <https://localhost:8080/release>) wechseln und in *PIX on Windows* können nun mit einem Klick auf das Kamerasymbol *frames* erfasst werden. Nach kurzer Zeit erscheinen dann die erfassten *frames* in einer Liste darunter und können dann detailliert angeschaut werden (siehe Abbildung 4.3).

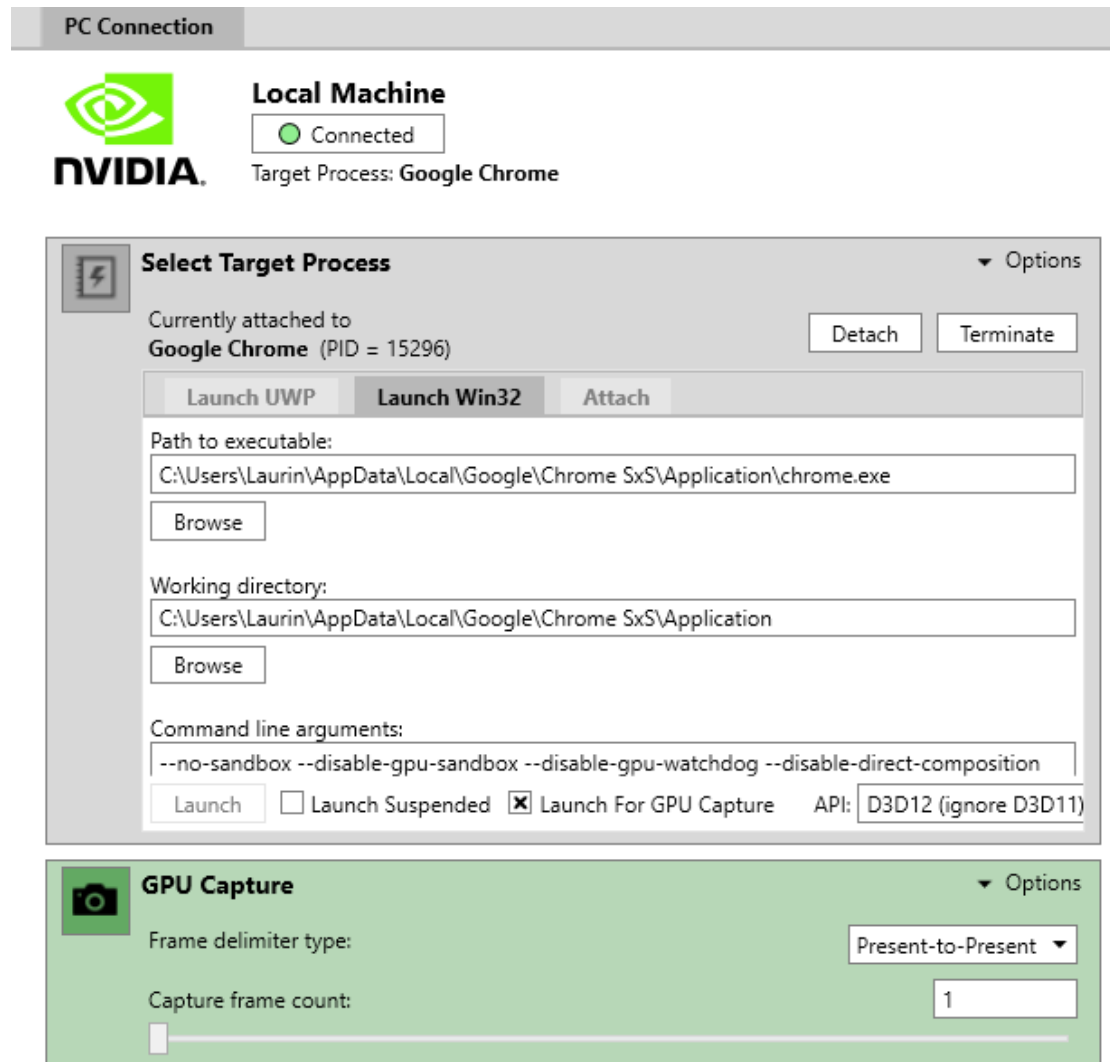


Abbildung 4.2.: Übersicht der Optionen zum Starten von *Google Chrome Canary* in *Microsoft PIX on Windows* [26]. Bildschirmaufnahme durch Verfasser.

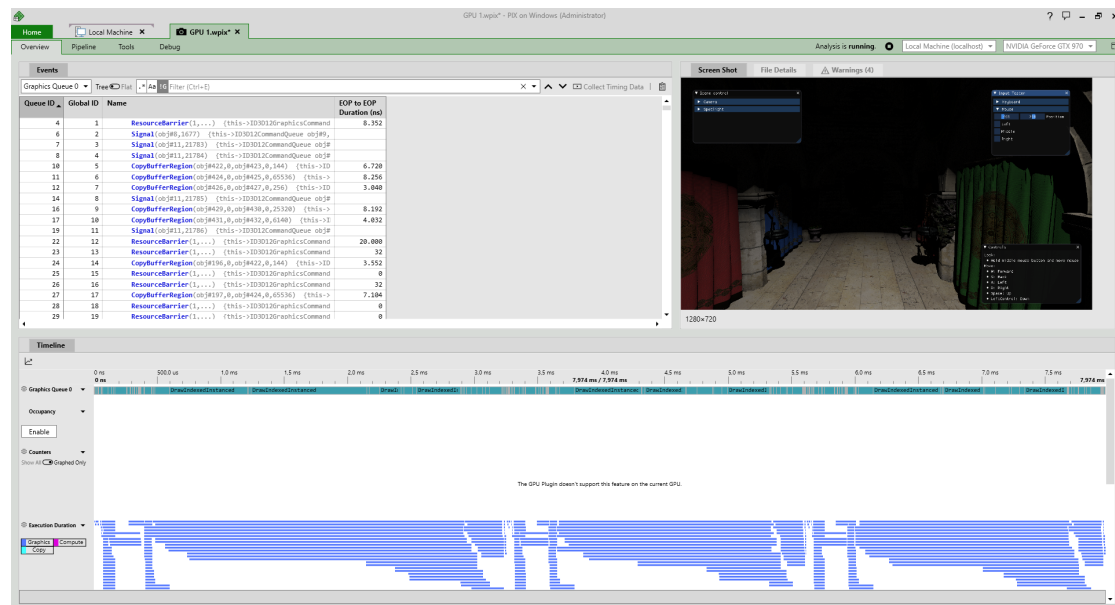


Abbildung 4.3.: Anzeige nach der Erfassung eines Einzelbildes in *Microsoft PIX on Windows* [26]. Bildschirmaufnahme durch Verfasser.

5. Evaluierung

6. Zusammenfassung und Ausblick

6.1. Erreichte Ergebnisse

6.2. Ausblick

6.2.1. Erweiterbarkeit der Ergebnisse

6.2.2. Übertragbarkeit der Ergebnisse

Referenzen

- [1] Advanced Micro Devices, Inc. *AMD Ryzen™ Threadripper™ 3990X*. Hardware. URL: <https://www.amd.com/de/products/cpu/amd-ryzen-threadripper-3990x> (besucht am 03.06.2020).
- [2] Tomas Akenine-Möller, Eric Haines und Naty Hoffman. *Real-Time Rendering*. Fourth Edition. Boca Raton, Fla: CRC Press, 2018. ISBN: 978-1-351-81615-1.
- [3] Robert L. Cook. „Shade Trees“. In: *Computer Graphics*. Bd. 18. 3. 1984.
- [4] Omar Cornut und Dear ImGui community. *Dear ImGui*. Software (API). URL: <https://github.com/ocornut/imgui> (besucht am 14.06.2020).
- [5] Emscripten Contributors (<https://emscripten.org/docs/contributing/AUTHORS.html>). *Emscripten*. Software (Werkzeug). URL: <https://emscripten.org/> (besucht am 15.06.2020).
- [6] Emscripten Contributors (<https://emscripten.org/docs/contributing/AUTHORS.html>). *Getting Started*. Software (Werkzeug). URL: https://emscripten.org/docs/getting_started/index.html (besucht am 15.06.2020).
- [7] Google Corporation. *Chrome Canary*. Software (Anwendung). URL: <https://www.google.com/intl/de/chrome/canary/> (besucht am 12.06.2020).
- [8] Google Corporation. *Chromium*. Software (Anwendung). URL: <https://www.chromium.org/Home> (besucht am 15.06.2020).
- [9] Google Corporation. *Dawn, a WebGPU implementation*. Software (Implementierung). URL: <https://dawn.googlesource.com/dawn> (besucht am 15.06.2020).
- [10] GPU for the Web Community Group. *GPU for the Web Community Group*. URL: <https://www.w3.org/community/gpu/> (besucht am 20.06.2020).
- [11] GPU for the Web Community Group. *Implementation Status*. URL: <https://github.com/gpuweb/gpuweb/wiki/Implementation-Status> (besucht am 09.06.2020).
- [12] GPU for the Web Community Group. *WebGPU*. Hrsg. von Dzmitry Malyshev, Justin Fan und Kai Ninomiya. Editor's Draft, 8 June 2020. URL: <https://gpuweb.github.io/gpuweb/> (besucht am 09.06.2020).
- [13] GPU for the Web Community Group. *WebGPU Shading Language*. Hrsg. von Dan Sinclair und Myles C. Maxfield. Editor's Draft, 22 June 2020. URL: <https://gpuweb.github.io/gpuweb/wgsl.html> (besucht am 20.06.2020).

- [14] Sebastian Grüner. *Apple stellt moderne 3D-Grafik für das Web vor*. URL: <https://www.golem.de/news/webgpu-apple-stellt-moderne-3d-grafik-fuer-das-web-vor-1702-126075.html> (besucht am 20.06.2020).
- [15] Intel Corporation. *Intel® Core™ i3 Prozessor 9100*. Hardware. URL: <https://ark.intel.com/content/www/de/de/ark/products/134870/intel-core-i3-9100-processor-6m-cache-up-to-4-20-ghz.html> (besucht am 03.06.2020).
- [16] Intel Corporation. *Intel® Core™ i9-9900K Prozessor*. Hardware. URL: <https://ark.intel.com/content/www/de/de/ark/products/186605/intel-core-i9-9900k-processor-16m-cache-up-to-5-00-ghz.html> (besucht am 03.06.2020).
- [17] Joshua-Ashton. *WGSL is terrible!* Online-Diskussion. URL: <https://github.com/gpuweb/gpuweb/issues/566> (besucht am 20.06.2020).
- [18] Kangz. *Consider adding sugar for familiar loop constructs*. Online-Diskussion. URL: <https://github.com/gpuweb/gpuweb/issues/569> (besucht am 20.06.2020).
- [19] Baldur Karlsson und RenderDoc community. *RenderDoc*. Software (Anwendung). URL: <https://renderdoc.org> (besucht am 12.06.2020).
- [20] Khronos Group. *glTF Overview*. Datenformat. URL: <https://www.khronos.org/glTF/> (besucht am 15.06.2020).
- [21] Khronos Group. *Sponza*. 3D-Modell. Original Crytek Sponza model provided at: <http://www.crytek.com/cryengine/cryengine3/downloads> glTF conversion provided by @Themaister at: <http://themaister.net/sponza-glTF-pbr/> There is no explicitly stated license for the model. The source page states: „The Atrium Sponza Palace, Dubrovnik, is an elegant and improved model created by Frank Meinl. The original Sponza model was created by Marko Dabrovic in early 2002. Over the years, the Sponza Atrium scene has become one of the most popular 3D scenes for testing global illumination and radiosity due to it's specific architectural structure which is particularly complex for global illumination light. However, nowadays it is considered as a simple model, thus it was decided to crate a new model with highly improved appearance and scene complexity. It is donated to the public for radiosity and is represented in several different formats (3ds, Obj) for use with various commercial 3D applications and renderers.“ URL: <https://github.com/KhronosGroup/glTF-Sample-Models/tree/master/2.0/Sponza> (besucht am 15.06.2020).
- [22] learnopengl.com. *Overview of coordinate systems*. Bild. URL: <https://learnopengl.com/Getting-started/Coordinate-Systems> (besucht am 20.06.2020).
- [23] LLVM community. *LLVM*. Software (Werkzeug). URL: <https://llvm.org/> (besucht am 15.06.2020).

- [24] Chris McClanahan. *History and Evolution of GPU Architecture*. Survey Paper. 2010. URL: <http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/94-CUDA/Docs/gpu-hist-paper.pdf>.
- [25] Microsoft Corporation. *D3D11On12*. Software (API). URL: <https://github.com/microsoft/D3D11On12> (besucht am 12.06.2020).
- [26] Microsoft Corporation. *PIX on Windows*. Software (Anwendung). URL: <https://devblogs.microsoft.com/pix/> (besucht am 12.06.2020).
- [27] Microsoft Corporation. *What is the Windows Subsystem for Linux?* Software (Werkzeug). URL: <https://docs.microsoft.com/de-de/windows/wsl/about> (besucht am 15.06.2020).
- [28] Mozilla Foundation. *Firefox Nightly*. Software (Anwendung). URL: <https://www.mozilla.org/de/firefox/all/#product-desktop-nightly> (besucht am 12.06.2020).
- [29] Mozilla Foundation. *wgpu*. Software (Implementierung). URL: <https://hg.mozilla.org/mozilla-central/file/tip/gfx/wgpu> (besucht am 15.06.2020).
- [30] NVIDIA Corporation. *NVIDIA GeForce GTX 1650*. Hardware. URL: <https://www.nvidia.com/de-de/geforce/graphics-cards/gtx-1650/> (besucht am 03.06.2020).
- [31] NVIDIA Corporation. *NVIDIA GeForce RTX 2080Ti*. Hardware. URL: <https://www.nvidia.com/de-de/geforce/graphics-cards/rtx-2080-ti> (besucht am 03.06.2020).
- [32] NVIDIA Corporation. *NVIDIA® Nsight™ Graphics*. Software (Anwendung). URL: <https://developer.nvidia.com/nsight-graphics> (besucht am 12.06.2020).
- [33] Online users. *Discussion on: WebGPU Shading Language*. Online-Diskussion. URL: <https://news.ycombinator.com/item?id=22536961> (besucht am 20.06.2020).
- [34] Pixar. *RenderMan Shading Language*. Spezifikation. URL: https://renderman.pixar.com/resources/RenderMan_20/shadingLanguage.html (besucht am 12.06.2020).
- [35] Scratchapixel. *Figure 3: barycentric coordinates can be used to interpolate vertex data at the hit point. In this example for example we compute the color at P using vertex color*. Bild. URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates> (besucht am 20.06.2020).
- [36] shaderc Contributors. *shaderc*. Software (Werkzeug). URL: <https://github.com/google/shaderc> (besucht am 15.06.2020).
- [37] WebAssembly community. *WebAssembly*. Spezifikation. URL: <https://webassembly.org/> (besucht am 15.06.2020).

- [38] Andre Weissflog. *Handles are the better pointers*. URL: <https://flooh.github.io/2018/06/17/handles-vs-pointers.html> (besucht am 15.06.2020).
- [39] Andre Weissflog. *Modern C for C++ Peeps*. URL: <https://flooh.github.io/2019/09/27/modern-c-for-cpp-peeps.html> (besucht am 15.06.2020).
- [40] Wikimedia Commons. *Picture of a Zotax Gaming 2080 ti Graphics Card*. Bild. 2019. URL: https://commons.wikimedia.org/wiki/File:Zotac_Gaming_GTX_2080_ti.jpg (besucht am 03.06.2020).
- [41] Wikipedia, the free encyclopedia. *3dfx Voodoo Graphics*. URL: https://de.wikipedia.org/wiki/3dfx_Voodoo_Graphics (besucht am 20.06.2020).
- [42] Wikipedia, the free encyclopedia. *GeForce 256*. URL: https://en.wikipedia.org/wiki/GeForce_256 (besucht am 20.06.2020).
- [43] Wikipedia, the free encyclopedia. *GeForce 3 series*. URL: https://en.wikipedia.org/wiki/GeForce_3_series (besucht am 20.06.2020).
- [44] Wikipedia, the free encyclopedia. *High-Level Shading Language*. URL: https://en.wikipedia.org/wiki/High-Level_Shading_Language (besucht am 20.06.2020).
- [45] Wikipedia, the free encyclopedia. *Swizzling (computer graphics)*. URL: [https://en.wikipedia.org/wiki/Swizzling_\(computer_graphics\)](https://en.wikipedia.org/wiki/Swizzling_(computer_graphics)) (besucht am 20.06.2020).
- [46] Wikipedia, the free encyclopedia. *WebGPU*. URL: <https://en.wikipedia.org/wiki/WebGPU> (besucht am 20.06.2020).

A. Anhang A

Dies ist der Quellcode zu der in Abbildung 4.1 gezeigten Beispielanwendung. Die Anwendung lädt dabei das relativ komplexe 3D-Modell **Sponza** [21] aus einer glTF-Datei [20], lässt den/die Anwender*in sich mit Hilfe einer steuerbaren Kamera im Raum bewegen und durch das UserInterface verschiedene Parameter der Szene verändern.

```
1 #include "spider/spider.h"
2
3 static SPLightID spot_light_id;
4 static uint32_t last_mouse_pos_x = 0;
5 static uint32_t last_mouse_pos_y = 0;
6 static const uint32_t surface_width = 1280;
7 static const uint32_t surface_height = 720;
8 static vec3 cam_rot = {0.0f, 0.0f, 0.0f};
9 static vec4 forward = {0.0f, 0.0f, 1.0f, 0.0f};
10 static float sensitivity = 2.0f;
11 static float vertical_limit = 0.01f;
12
13 void init(void) {
14     // Lights have to be created before materials right now
15     const vec3 light_pos = {0.0f, 5.0f, 0.5f};
16     const vec3 light_look_at = {2.0f, 0.0f, 0.0f};
17     vec3 light_direction = {-1.0, -1.0f, 0.2f};
18     // (float*) cast to prevent compiler warning
19     // cglm takes no const pointers as arguments, even if it doesn't mutate the
20     // vectors
21     glm_vec3_sub((float*)light_look_at, (float*)light_pos, light_direction);
22     glm_vec3_normalize(light_direction);
23
24     spot_light_id = spCreateSpotLight(&(SPSpotLightDesc){
25         .pos = {light_pos[0], light_pos[1], light_pos[2]},
26         .range = 40.0f,
27         .color = {.r = 255, .g = 255, .b = 255},
28         .dir = {light_direction[0], light_direction[1], light_direction[2]},
29         .fov = glm_rad(70.0f),
30         .power = 20.0f,
31         .shadow_casting = &(SPLightShadowCastDesc){
32             .shadow_map_size = 2048,
33         },
34     });
35     SP_ASSERT(spot_light_id.id != SP_INVALID_ID);
36
37     /*SPSceneNodeID sponza_node_id = */spLoadGltf("assets/gltf/Sponza/Sponza.gltf");
38 }
39
40 bool update(float delta_time_s) {
41     static bool show_controls = true;
```



```

42     igBegin("Controls", &show_controls, ImGuiWindowFlags_None);
43         igText("Look:");
44         igBulletText("Hold right mouse button and move mouse");
45         igText("Move:");
46         igBulletText("W: Forward");
47         igBulletText("S: Back");
48         igBulletText("A: Left");
49         igBulletText("D: Right");
50         igBulletText("Space: Up");
51         igBulletText("LeftControl: Down");
52     igEnd();
53
54     static bool scene_control = true;
55     igBegin("Scene control", &scene_control, ImGuiWindowFlags_None);
56
57     SPCamera* cam = spGetActiveCamera();
58     glm_vec4_normalize(forward);
59
60     if(cam) {
61         if(spGetMouseButtonPressed(SPMouseButton_Right)) {
62             vec2 relative_delta = {
63                 ((float)spGetMousePositionX() - (float)last_mouse_pos_x) / (float)
                    surface_width,
64                 ((float)spGetMousePositionY() - (float)last_mouse_pos_y) / (float)
                    surface_height
65             };
66             float rotation_speed = sensitivity * M_PI;
67             cam_rot[1] -= rotation_speed * relative_delta[0]; // horizontal
68             cam_rot[0] += rotation_speed * relative_delta[1]; // vertical
69             cam_rot[0] = glm_clamp(cam_rot[0], (-M_PI * 0.5f) + vertical_limit,
                (M_PI * 0.5f) - vertical_limit);
70         }
71         memcpy(forward, (vec4){0.0f, 0.0f, 1.0f, 0.0f}, sizeof(vec4));
72         mat4 rot = GLM_MAT4_IDENTITY_INIT;
73         glm_euler_zyx(cam_rot, rot);
74         glm_mat4_mulv(rot, forward, forward);
75
76         cam->dir[0] = forward[0];
77         cam->dir[1] = forward[1];
78         cam->dir[2] = forward[2];
79         vec3 sideward = {
80             -forward[2],
81             0.0f,
82             forward[0],
83         };
84         glm_vec3_normalize(sideward);
85         const float walk_speed = 2.0f;
86         const float forward_movement = walk_speed * delta_time_s * (-1.0f *
            spGetKeyPressed(SPKey_S) + spGetKeyPressed(SPKey_W));
87         const float sideward_movement = walk_speed * delta_time_s * (-1.0f *
            spGetKeyPressed(SPKey_A) + spGetKeyPressed(SPKey_D));
88         const float upward_movement = walk_speed * delta_time_s * (-1.0f *
            spGetKeyPressed(SPKey_ControlLeft) + spGetKeyPressed(SPKey_Space));
89         cam->pos[0] += forward[0] * forward_movement + sideward[0] *
            sideward_movement;
90         cam->pos[1] += forward[1] * forward_movement + upward_movement;
91         cam->pos[2] += forward[2] * forward_movement + sideward[2] *
            sideward_movement;
92
93         if(igCollapsingHeaderTreeNodeFlags("Camera", ImGuiTreeNodeFlags_None)) {
94             igSliderFloat("Look sensitivity##cam", &sensitivity, 0.0f, 5.0f,
                "%.1f", 1.0f);

```

```

95         igSliderFloat("Vertical look limit##cam", &vertical_limit, 0.0f, M_PI *
96             0.5f, "%.2f", 1.0f);
97         igSliderFloat3("Position##cam", (float*)&cam->pos, -10.0f, 10.0f,
98             "%.2f", 1.0f);
99         igSliderFloat3("Rotation (Rad)##cam", (float*)&cam_rot, -M_PI, M_PI,
100             "%.2f", 1.0f);
101         igSliderFloat("Vertical field of view (Rad)##cam", &cam->fovy, 0.01f,
102             M_PI, "%.2f", 1.0f);
103     }
104 }
105
106 last_mouse_pos_x = spGetMousePositionX();
107 last_mouse_pos_y = spGetMousePositionY();
108
109 SPLight* spot_light = spGetLight(spot_light_id);
110 if(spot_light) {
111     if(igCollapsingHeaderTreeNodeFlags("Spotlight", ImGuiTreeNodeFlags_None)) {
112         igSliderFloat3("Position##light", (float*)&spot_light->pos, -50.0f,
113             50.0f, "%.1f", 1.0f);
114         igSliderFloat("Field of view##light", &spot_light->fovy, 0.0f, M_PI,
115             "%.2f", 1.0f);
116         igSliderFloat("Power##light", &spot_light->power, 0.0f, 1000.0f,
117             "%.0f", 1.0f);
118         igSliderFloat("Range##light", &spot_light->range, 0.0f, 1000.0f,
119             "%.0f", 1.0f);
120     }
121 }
122 igEnd();
123
124 // return false if you want to quit
125 return true;
126 }
127
128 int main() {
129     spInit(&(SPInitDesc){
130         .surface_size = {
131             .width = surface_width,
132             .height = surface_height
133         },
134         .update_func = update,
135         .camera = {
136             .pos = {0.0f, 2.0f, 0.0f},
137             .dir = {0.0f, 0.0f, 1.0f},
138             .look_at = {0.0f, 0.0f, 0.0f},
139             .mode = SPCameraMode_Direction,
140             .fovy = glm_rad(60.0f),
141             .aspect = (float)surface_width / (float)surface_height,
142             .near = 0.1f,
143         },
144         .pools.capacities = {
145             .meshes = 128,
146             .materials = 64,
147             .render_meshes = 256,
148             .lights = 1,
149             .scene_nodes = 1024,
150         },
151         .show_stats = true,
152     });
153
154     init();
155
156     spStart();

```

```
149     return 0;  
150 }
```

Listing A.1: Kompletter C99-Quellcode zur Erstellung einer interaktiven 3D-Szene mit der **spider**-Engine

B. Anhang B

```
1 #version 450
2 #extension GL_ARB_separate_shader_objects : enable
3 #extension GL_EXT_samplerless_texture_functions : enable
4
5 layout(set = 0, binding = 1) uniform Camera {
6     mat4 view;
7     mat4 proj;
8     vec3 pos;
9 } cam;
10
11 layout(set = 0, binding = 2) uniform Light {
12     mat4 view;
13     mat4 proj;
14     vec4 pos3_rangel;
15     vec4 color3_type1; // type: 0 directional, 1 spot, 2 point
16     vec4 dir3_fov1; // dir: for spot & dir, fov: for spot
17     vec4 area2_power1_padding1; // area: for dir
18 } light; // TODO: support more than 1 light
19
20 layout(set = 2, binding = 0) uniform texture2D albedo_tex;
21 layout(set = 2, binding = 1) uniform sampler albedo_sampler;
22 layout(set = 2, binding = 2) uniform texture2D normal_tex;
23 layout(set = 2, binding = 3) uniform sampler normal_sampler;
24 layout(set = 2, binding = 4) uniform texture2D ao_roughness_metallic_tex;
25 layout(set = 2, binding = 5) uniform sampler arm_sampler;
26 layout(set = 2, binding = 6) uniform texture2D shadow_map;
27 layout(set = 2, binding = 7) uniform sampler shadow_sampler;
28
29
30 layout(location = 0) in vec3 fragPosWorld;
31 layout(location = 1) in vec2 fragTexCoords;
32 layout(location = 2) in vec3 fragNormal;
33 layout(location = 3) in vec3 fragTangent;
34
35 layout(location = 0) out vec4 outColor;
36
37 const float PI = 3.14159265359;
38 const float gamma = 2.2;
39
40 const int sampleSize = 1;
41 const float scale = 1.5;
42
43 float getLightDepthOnPosSingle(vec2 coords_shadow_map) {
44     return texture(sampler2D(shadow_map, shadow_sampler), coords_shadow_map.xy)
45         .r;
46 }
47
48 float getLightDepthOnPosSampled(vec2 coords_shadow_map) {
49     float value = 0.0;
50     ivec2 texDim = textureSize(shadow_map, 0);
51     float dx = scale * 1.0 / float(texDim.x);
```

```

51     float dy = scale * 1.0 / float(texDim.y);
52     int count = 0;
53     for(int y = -sampleSize; y <= sampleSize; ++y) {
54         for(int x = -sampleSize; x <= sampleSize; ++x) {
55             value += getLightDepthOnPosSingle(coords_shadow_map + vec2(x * dx, y *
56                 dy));
57             count++;
58         }
59     }
60     return value / count;
61 }
62 float DistributionGGX(vec3 N, vec3 H, float roughness) {
63     float a = roughness*roughness;
64     float a2 = a*a;
65     float NdotH = max(dot(N, H), 0.0);
66     float NdotH2 = NdotH*NdotH;
67
68     float num = a2;
69     float denom = (NdotH2 * (a2 - 1.0) + 1.0);
70     denom = PI * denom * denom;
71
72     return num / denom;
73 }
74
75 float GeometrySchlickGGX(float NdotV, float roughness) {
76     float r = (roughness + 1.0);
77     float k = (r*r) / 8.0;
78
79     float num = NdotV;
80     float denom = NdotV * (1.0 - k) + k;
81
82     return num / denom;
83 }
84 float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness) {
85     float NdotV = max(dot(N, V), 0.0);
86     float NdotL = max(dot(N, L), 0.0);
87     float ggx2 = GeometrySchlickGGX(NdotV, roughness);
88     float ggx1 = GeometrySchlickGGX(NdotL, roughness);
89
90     return ggx1 * ggx2;
91 }
92
93 vec3 fresnelSchlick(float cosTheta, vec3 F0) {
94     return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
95 }
96
97 void main() {
98     const vec4 albedo_all = texture(sampler2D(albedo_tex, albedo_sampler),
99         fragTexCoords).rgba;
100     const vec3 albedo = albedo_all.rgb;
101     const float alpha = albedo_all.a;
102     if(alpha == 0.0) {
103         discard;
104     }
105     vec3 tangent_space_normal = normalize(texture(sampler2D(normal_tex,
106         normal_sampler), fragTexCoords).xyz * vec3(2.0, 2.0, 1.0) - vec3(1.0, 1.0,
107         0.0));
108     const vec3 ao_roughness_metallic = texture(sampler2D(ao_roughness_metallic_tex,
109         arm_sampler), fragTexCoords).rgb;
110     const float ao = ao_roughness_metallic.r;
111     const float roughness = ao_roughness_metallic.g;

```

```

108     const float metallic      = ao_roughness_metallic.b;
109
110     vec3 Normal = normalize(fragNormal);
111     vec3 Tangent = normalize(fragTangent);
112     // enforce orthogonality
113     Tangent = normalize(Tangent - dot(Tangent, Normal) * Normal);
114     const vec3 Bitangent = cross(Tangent, Normal);
115     mat3 TBN = mat3(Tangent, Bitangent, Normal);
116     vec3 normal = normalize(TBN * tangent_space_normal);
117
118     vec3 N = normalize(normal);
119     vec3 V = normalize(cam.pos - fragPosWorld);
120     vec3 F0 = vec3(0.04);
121     F0 = mix(F0, albedo, metallic);
122
123     // reflectance equation
124     vec3 Lo = vec3(0.0);
125     vec3 light_pos = light.pos3_range1.xyz;
126     float distance = length(light_pos - fragPosWorld);
127     vec3 L = normalize(light_pos - fragPosWorld);
128     float light_fov = 1.0 - (light.dir3_fov1.w / 3.14);
129     float attenuation = light.area2_power1_padding1.z * (light_fov * light_fov);
130     attenuation *= max(1.0 - distance / light.pos3_range1.w, 0.0);
131
132     // For spot lights
133     if(light.color3_type1.w == 1.0) {
134         const float light_angle_rad = acos(dot(light.dir3_fov1.xyz, -L));
135         attenuation *= pow(max(light.dir3_fov1.w * 0.5 - light_angle_rad, 0.0) /
136             3.14, 1.0 - (light_fov * light_fov));
137     }
138     // Shadow calculation
139     const vec4 pos_shadow_map = light.proj * light.view * vec4(fragPosWorld, 1.0);
140     vec4 pos_in_light_clip_space = pos_shadow_map / pos_shadow_map.w;
141     pos_in_light_clip_space.xy = pos_in_light_clip_space.xy * 0.5 + 0.5; // [-1,1]
142     // to [0,1]
143     pos_in_light_clip_space.y = 1.0 - pos_in_light_clip_space.y; // bottom-up to
144     // top-down
145     // [0, 0] of pos_in_light_clip_space.xy should now be the top left corner and
146     // [1, 1] the bottom right --> texture space
147
148     const float depth_bias = 0.00001; // Depth bias is not yet implemented in
149     // Chromium/dawn, so we have to "fake" it in the shader
150     if(attenuation > 0.0 && pos_in_light_clip_space.z > 0.0 &&
151         pos_in_light_clip_space.z < 1.0) {
152         const float light_depth_on_pos =
153             getLightDepthOnPosSampled(pos_in_light_clip_space.xy);
154         if(pos_in_light_clip_space.w > 0.0 && light_depth_on_pos - depth_bias >
155             pos_in_light_clip_space.z) {
156             attenuation = 0.0;
157         }
158     }
159     if(attenuation > 0.0) {
160         // calculate per-light radiance
161         vec3 H = normalize(V + L);
162         vec3 radiance = light.color3_type1.rgb * attenuation;
163
164         // cook-torrance brdf
165         float NDF = DistributionGGX(N, H, roughness);
166         float G = GeometrySmith(N, V, L, roughness);
167         vec3 F = fresnelSchlick(max(dot(H, V), 0.0), F0);
168
169         vec3 kS = F;

```

```
162     vec3 kD = vec3(1.0) - kS;
163     kD *= 1.0 - metallic;
164
165     vec3 numerator      = NDF * G * F;
166     float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0);
167     vec3 specular      = numerator / max(denominator, 0.001);
168
169     // add to outgoing radiance Lo
170     float NdotL = max(dot(N, L), 0.0);
171     Lo += (kD * albedo / PI + specular) * radiance * NdotL;
172 }
173
174 vec3 ambient = vec3(0.04) * albedo * ao;
175 vec3 color = ambient + Lo;
176
177 color = color / (color + vec3(1.0));
178 color = pow(color, vec3(1.0/gamma));
179
180 outColor = vec4(color, alpha);
181 }
```

Listing B.1: Kompletter GLSL-Code des Fragment-Shaders der in der **spider-Engine** für das PBR benutzt wird