Hochschule Aalen

Bachelorarbeit
Studiengang Medieninformatik

WebGPU

von

Laurin Agostini

60526

Betreuender Professor:	Prof. Dr. Winfried Bantel
Zweitprüfer:	Prof. Dr. Carsten Lecon
Einreichungsdatum:	29. 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

Mit dem schleichenden Wechsel der Webbrowsern von reinen Anzeigewerkzeugen für einfache Webseiten hin zu einer Laufzeitumgebung mit integrierter Softwaredistributionsplattform, werden die Anforderungen an die Webbrowser immer ähnlicher zu denen von Betriebssystemen. Ein wichtiger Teil moderner Applikationen ist dabei auch das effiziente Anzeigen und Erstellen von Grafiken. Zwar hat die Web-Plattform mit WebGL (bzw. WebGL 2.0) eine Grafik-API, die den meisten Anforderungen gerecht wird, jedoch ist auch bei den Grafik-APIs seit ungefähr Mitte des vergangenen Jahrzehnts ein Paradigmenwechsel von „einfach zu benutzen“ hin zu „mehr Kontrolle für die Entwickler“ zu beobachten. Um in diesem Gebiet aufzuschließen, wird die Grafik-API **WebGPU** entwickelt.

In dieser Arbeit werden dafür die Grundlagen von Grafik-APIs beschrieben um dann die Geschichte und Beweggründe hinter **WebGPU** zu schildern. Dann werden Aspekte der aktuell verfügbaren Version der **WebGPU**-API ausführlich besprochen und mit ihnen ein Minimalbeispiel zur Benutzung beschrieben und ausgearbeitet. An diesem lässt sich dann gut erkennen, dass die Einstiegshürde zur Erstellung einer Applikation relativ hoch liegt und sich sehr von den derzeitig im Web verfügbaren Grafik-APIs unterscheidet. **WebGPU** lehnt sich dabei in ihrer Struktur und Funktionsweise sehr an die „modernen“ Grafik-APIs Vulkan, DirectX 12 und Metal an, schafft es aber trotzdem, die Komplexität im Vergleich niedriger zu halten. Deswegen wird auch auf die Möglichkeit eingegangen, **WebGPU** als Grafik-API für zukünftige Cross-Platform-Applikationen zu verwenden. Dies wird exemplarisch mit der parallel zu dieser Arbeit entstandenen **spider**-Engine aufgezeigt. Die **spider**-Engine bildet dabei ein C-Framework, dass es mithilfe des Compiler-Werkzeugs emscripten erlaubt, relativ einfach 3D-Applikationen in C zu schreiben aber dann später als Webseite zu veröffentlichen.

Als Ergebnis dieser Arbeit stellt sich heraus, dass die **WebGPU**-API einen guten Kompromiss zwischen der Komplexität und dem tatsächlichem Nutzen der bestehenden Desktop-Grafik-APIs findet. Ob sich **WebGPU** aber langfristig großer Popularität erfreuen kann, liegt, wie bei den meisten Web-APIs, an der Verfügbarkeit und Unterstützung in den Webbrowsern. Hier liegen zwar experimentelle Implementierungen vor, diese ändern sich (wie Teile der Spezifikation) aber wöchentlich, so dass der produktive Einsatz noch nicht greifbar scheint.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

CPU Central Processing Unit (dt. Hauptprozessor) 3
GPU Graphics Processing Unit (dt. Grafikprozessor) 3
SMT Simultaneous Multithreading 3
PCIe Peripheral Component Interconnect Express 4
API Application Programming Interface (dt. Programmierschnittstelle) 47
WSL Windows Subsystem for Linux 41
GLSL OpenGL Shading Language viii
HLSL High Level Shading Language 14
PBR Physically Based Rendering viii
AMD Advanced Micro Devices, Inc 14

1. Einleitung

1.1. Motivation

Wo Webseiten vor einem Jahrzehnt noch großteils der Informationsdarstellung dienten, wandern in den letzten Jahren immer mehr traditionelle Desktop-Applikationen in den Webbrowser. So ist zum Beispiel die ganze Office-Suite von Microsoft als WebApp im Webbrowser nutzbar [28]. Dies hilft natürlich extrem bei der Verbreitung von Applikationen, da diese nicht mehr aufwendig heruntergeladen und installiert werden müssen.

Anwendungen mit hohen Grafikanforderungen, wie Videospiele, Architektursoftware oder 3D-Visualisierungen, haben allerdings bisher diesen Schritt zu großen Teilen nicht geschafft. Zwar gibt es Grafik-APIs fürs Web, doch diese stehen hinter den nativen Desktop-Varianten im Bezug auf Umfang und Effizienz. Mit **WebGPU** soll nun dafür eine „moderne“ Grafik-API entwickelt werden, damit mehr Anwendungen im Web umgesetzt werden können.

1.2. Problemstellung und -abgrenzung

Da sich **WebGPU** sowohl bei der Spezifikation, als auch bei der tatsächlichen Implementierung in den Webbrowsern, noch in einem experimentellen Status befindet, soll diese Arbeit nur einen Überblick über die prinzipiellen Strukturen und Funktionsweisen geben. Auch der Vergleich mit anderen Grafik-APIs kann hier nur auf struktureller Ebene erfolgen, da zum Beispiel Performance- und Effizienzvergleiche aufgrund des experimentellen Status nicht aussagekräftig sind. Auch auf Funktionen abseits von der Kernfunktion einer Grafik-API, die Berechnung und Darstellung von Grafiken, wird nur am Rande eingegangen. Dies liegt daran, dass Funktionen wie *GPU-Compute* zur Berechnung „beliebiger Daten“, kaum oder nur unzureichend zum jetzigen Zeitpunkt in den Webbrowsern implementiert ist.

1.3. Ziel der Arbeit

Am Ende der Arbeit soll ein Überblick über **WebGPU** im Bezug auf Funktionsweise und noch bestehenden Schwächen vorhanden sein. Die Arbeit soll außerdem Interessierten an **WebGPU** einen Einstieg zur Erstellung einfacher Applikationen bieten.

1.4. Vorgehen

Der praktische Teil dieser Arbeit besteht aus dem Entwickeln eines C-Frameworks zur Erstellung von 3D-Web-Applikationen mithilfe des Compiler-Werkzeugs emscripten. Dadurch sollen praktische Erfahrungen mit der **WebGPU**-API und ihrer Unterstützung in Webbrowsern erlangt werden. Durch den offenen Entwicklungsprozess der Spezifikation und der Implementierungen, kann hier außerdem „hinter die Kulissen“ geschaut und eventuell sogar Feedback gegeben werden.

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-CPUs 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 [34]	4.352	1.350 / 1.545
Low-end		
Intel® Core™ i3-9100 [15]	4	3.600 / 4.200
NVIDIA GeForce GTX 1650 [33]	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) [45]

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-CPUs 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.

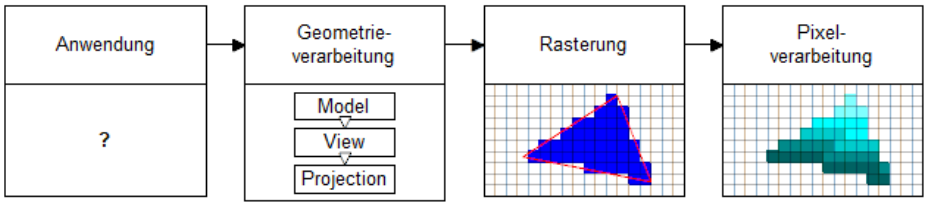

Abbildung 2.2.: Grundsätzlicher Aufbau der Rendering-Pipeline, unterteilt in die vier Abschnitte *Anwendung*, *Geometrieverarbeitung*, *Rasterung* und *Pixelverarbeitung*

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

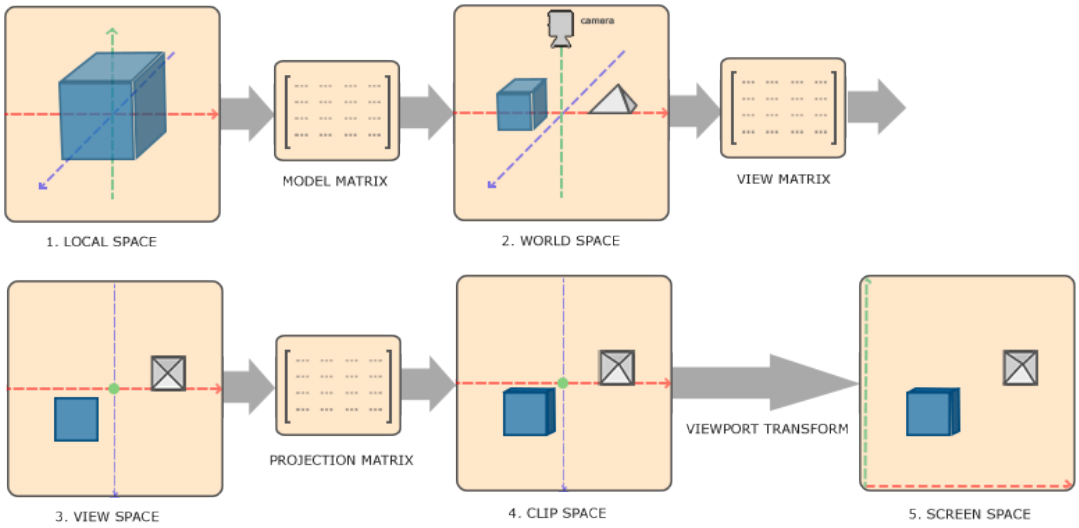

Abbildung 2.3.: Transformation von Objekt bezogenen Vertexdaten in den *world space*, dann in den *view space* und schlussendlich in den *clip space*. Die finale Umwandlung vom *clip space* in den *screen space* passiert dabei automatisch und verschiebt den Ursprung von der Mitte in entweder die ober oder untere (abhängig von der verwendeten Grafik-API) linke Ecke. [24]

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 Ausgansparameter 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 anderseits 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.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(set = 0, binding = 0) uniform Model {
    mat4 model;
};

layout(set = 0, binding = 1) uniform Camera {
    mat4 view;
    mat4 proj;
    vec3 pos;
} cam;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec2 inTexCoords;
layout(location = 2) in vec3 inNormal;
layout(location = 3) in vec3 inTangent;

layout(location = 0) out vec3 fragPosWorld;
layout(location = 1) out vec2 fragTexCoords;
layout(location = 2) out vec3 fragNormal;
layout(location = 3) out vec3 fragTangent;

void main() {
    vec3 pos_world = vec3(model * vec4(inPosition, 1.0));
    gl_Position = cam.proj * cam.view * vec4(pos_world, 1.0);
    fragPosWorld = pos_world;
    fragTexCoords = inTexCoords;
    mat3 to_world = mat3(transpose(inverse(model)));
    fragNormal = to_world * inNormal;
    fragTangent = to_world * inTangent;
}
```

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

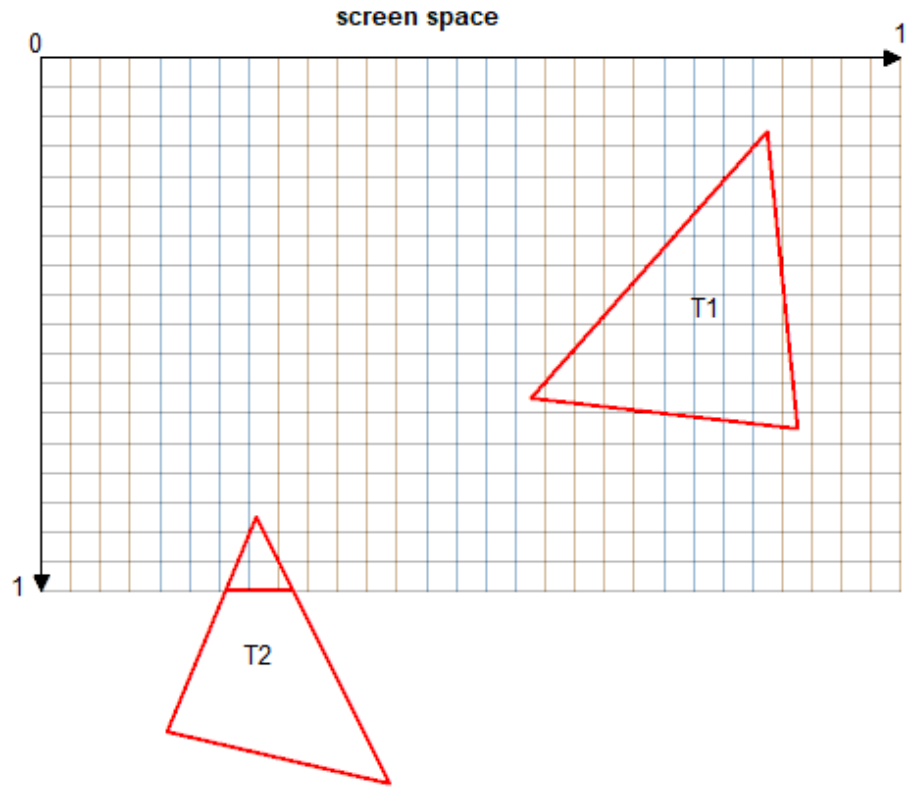

Abbildung 2.4.: Zwei Dreiecke **T1** und **T2** im *screen space*. **T1** wird hierbei komplett gerastert und **T2** erst so zerteilt, dass nur der sichtbare Bereich gerastert werden muss.

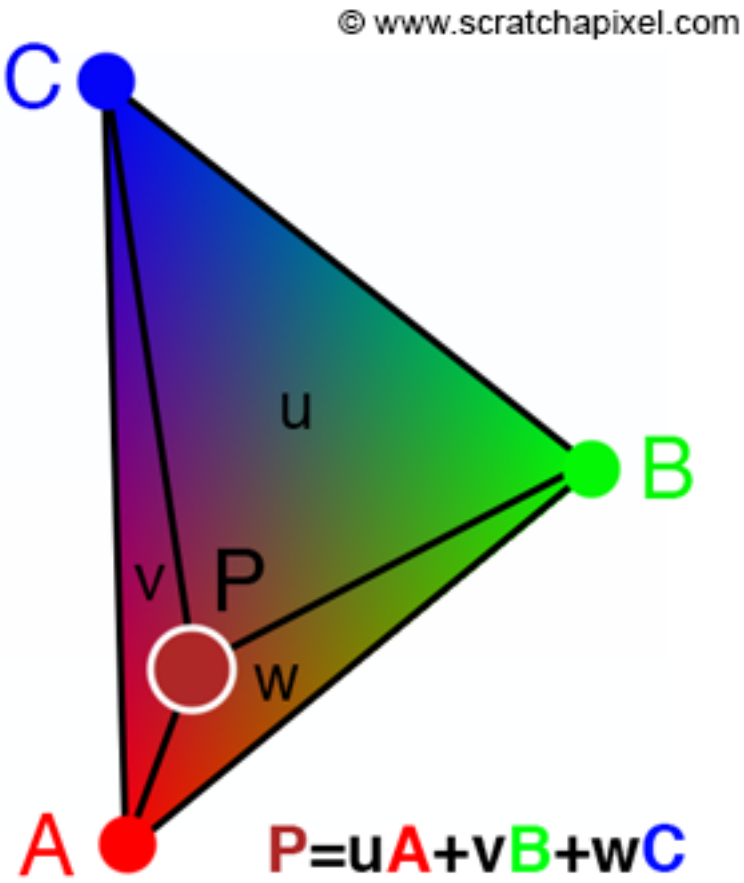

Abbildung 2.5.: Interpolation von Vertexdaten innerhalb eines Dreiecks am Beispiel der Farbinformation der Vertices mithilfe von baryzentrischen Koordinaten [38]

- Wie soll die Vorderseite eines Dreiecks bestimmt werden? Vertices sind entweder im oder gegen den Uhrzeigersinn angeordnet
- Welche Seiten eines Dreieckes 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 Ausgansparameter 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.

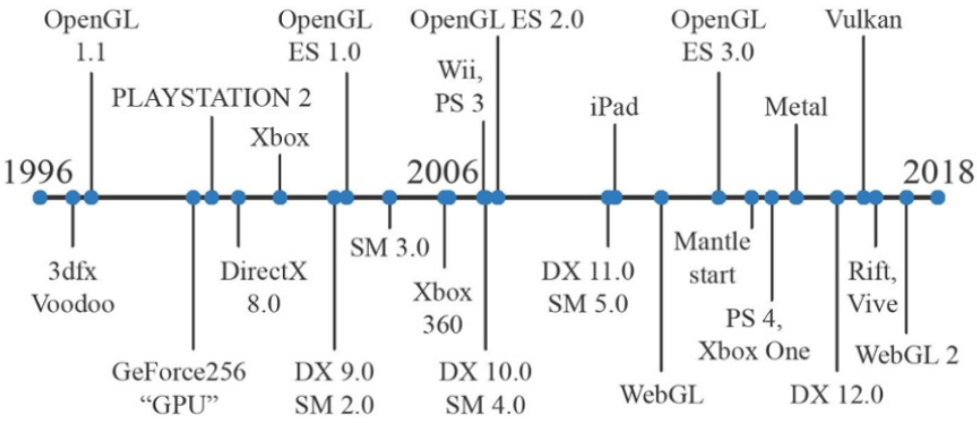

Abbildung 2.6.: Zeitlinie mit wichtigen Grafik-API- und Hardwareerscheinungen [2, S. 38]

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 [37] 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. 26].

Die erste GPU zur Beschleunigung von 3D-Szenen, die 3dfx Voodoo [46], hatte die Rendering-Pipeline direkt in die Hardware implementiert. Auch die NVIDIA GeForce 256 [47], 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 [48] 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. 50, 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 unterstütze 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) [50] für DirectX und GLSL [50] 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 Funktiona-

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. 57)

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 **WebGPU**-Implementierungen unterstützen jedoch bisher nur SPIR-V als Shader-Code. 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. Jedoch ist hier zu beachten, dass dies nur dem aktuellen Stand der Spezifikation entspricht und sich hier noch vieles verändern kann.

3.3.2. Kritik an WGSL

```
// for-loop in GLSL
int a = 2;
const int step = 1;
for (int i = 0; i < 4; i += step) {
  if (i % 2 == 0) continue;
  a *= 2;
}

// "for-loop" in WGSL
const a : i32 = 2;
var i : i32 = 0;
loop {
  if (i >= 4) { break; }

  const step : i32 = 1;

  if (i % 2 == 0) { continue; }

  a = a * 2;

  continuing {
    i = i + step;
  }
}
```

Listing 3.1: for-Schleifen in GLSL und 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 (siehe Listing 3.1), aber nicht direkt `for`, `while` oder `do-while` [Vgl. 19]. 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 etwas 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* [39] 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). Anderseits scheint es wohl rechtliche Spannungen zwischen Apple und der für SPIR-V verantwortlichen Khronos Group zu geben [Vgl. 36].

3.3.3. Vergleich zu GLSL und HLSL

In diesem Abschnitt sollen die Grundlagen von WGSL aufgezeigt werden, im Besonderen im Bezug auf Unterschiede zu GLSL und HLSL. Da sich die Spezifikation von WGSL jedoch noch in Arbeit befindet, ist alles folgende nur auf Grundlage der zur Zeit des Verfassens dieses Abschnittes (24. Juni 2020) aktuellen Spezifikation.

Skalare Typen

Typ	**Beschreibung**	**GLSL**	**HLSL**
Boolean			
`bool`	Entweder `true` oder `false`	`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

`vec`*N*`<`*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`vec`*N*

wobei *T* hier als Abkürzung eines skalaren Typen fungiert (b = `bool`, i = `int`, u = `uint`, d = `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:

*T*N

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

mat*N*x*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

mat*N*x*M*

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

mat*N*

Matrixkomponenten sind immer vom Typ `float` oder `double`. Eine Matrix mit doppelter Genauigkeit wird mit `dmat`*N*x*M* bzw `dmat`*N* deklariert.

In HLSL werden die Matrixtypen ähnlich wie die Vektortypen deklariert:

*T*NxM

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* [55] genannt wird (siehe Listing 3.2).

```
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 3.2: „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].

Variablen

Im Gegensatz zu den C ähnlichen Sprachen GLSL und HLSL, verwendet WGSL eine Syntax zur Variablendefinition, die häufig in „moderneren“ Sprachen wie Rust oder TypeScript benutzt wird:

```
var Name : Type [= InitialValue];
```

bzw.

```
const Name : Type = Value;
```

Wird der initiale Wert weggelassen, wird die Variable mit `0` (bzw. `0.0` / `false`) initialisiert [Vgl. 13, Zero values].

Strukturen

Eine Struktur ist ein Tupel mit N Felder, wobei N eine Ganzzahl größer `0` ist. Die Felder *T1* bis *TN* haben jeweils einen Datentyp, welcher wiederum eine Struktur sein kann (siehe Listing 3.3).

```
type Student = struct {
  grade : i32;
  GPA : f32;
  attendance : array<bool,4>;
};

# The zero value for Student
var student: Student = Student(2, 1.5, array<bool,4>(false,false,true,false));
```

Listing 3.3: Definition und Benutzung von Strukturen in WGSL

3.4. Der aktuelle Stand

3.4.1. Spezifikation

Unter `https://gpuweb.github.io/gpuweb/` [12] kann die aktuellste Version der Spezifikation eingesehen werden.

Zum Zeitpunkt des Verfassens dieses Abschnittes (24. Juni 2020) steht ein Großteil der **WebGPU**-Spezifikation fest. Jedoch gibt es noch immer Punkte bei denen noch keine Einigung gefunden werden konnte. Da der Großteil der Implementierung der **spider**-Engine vor Juni 2020 statt gefunden hat, werden die folgenden Beschreibungen und Erkenntnisse zur Zeit der Abgabe der Arbeit (oder insbesondere zu einem späteren Zeitpunkt) nicht mehr zwingend aktuell sein. An der grundlegenden Struktur oder den grundlegenden Mechaniken zur Verwendung von **WebGPU** sollte sich zwar nicht mehr allzu viel ändern, trotzdem sollte die folgenden Abschnitte nicht als direkte Vorlage zum Implementieren einer **WebGPU**-Applikation gesehen wird.

3.4.2. Aktuelle Unterstützung in Webbrowsern

Der aktuelle Stand der Implementierung in Webbrowsern kann unter `https://github.com/gpuweb/gpuweb/wiki/Implementation-Status` [11] eingesehen werden.

Der produktive Nutzen von **WebGPU** hängt in erster Linie von der Unterstützung in den Webbrowsern ab. Laut StatCounter [40] benutzten im Mai 2020 über 95% aller Nutzer einer der in Tabelle 3.2 aufgelisteten Webbrowser.

Webbrowser	**Marktanteil**
Chrome	63,93%
Safari	18,19%
Firefox	4,38%
Samsung Internet	3,28%
Edge Legacy	2,13%
UC Browser	2,00%
Opera	1,92%
Internet Explorer	1,40%

Tabelle 3.2.: Marktanteile der verbreitetsten Webbrowser im Mai 2020 laut StatCounter [40]

Wobei hier anzumerken ist, dass sowohl Chrome [49], Samsung Internet [53] und

Opera [52] auf dem Webbrowser Chromium basieren. Chromium, Safari und Firefox haben jeweils eine experimentelle **WebGPU**-Implementierung, jedoch nur in den jeweiligen Preview-Versionen der Webbrowser. Dies lässt jedoch trotzdem den Schluss zu, dass **WebGPU** für mindestens 90% der Nutzer in naher Zukunft zur Verfügung stehen wird.

Chromium (8) / Dawn (9)

Dawn ist eine Open-Source **WebGPU**-Implementierung, die in dem Open-Source Webbrowser Chromium verwendet wird.

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 (31) / wgpu (32)

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. Ähnlich wie *Dawn* benutzt *wgpu* die nativen Grafik-APIs zur Übermittlung der Befehle an die GPU. Jedoch baut *wgpu* dabei auf die, ebenfalls von *Mozilla Foundation* entwickelten, Hardwarebstraktionsschicht *gfx-hal* auf und nicht direkt auf die einzelnen nativen Grafik-APIs.

4. Minimalbeispiel einer WebGPU-Applikation

In den folgenden Abschnitten wird für Codebeispiele die in der Spezifikation benutzten JavaScript-Notation für Strukturen und Funktionen verwendet. Die Konvertierung von der JavaScript-Notationen in die jeweilige C-Notation (die im Begleitprojekt verwendet wurde) ist dabei relativ simpel.

Für ein Objekt in der JavaScript-Notation, zum Beispiel `GPUDevice`, gibt es ein Äquivalent in der C-Notation: `WGPUDevice`. Dabei ist zu beachten, dass `WGPUDevice` nur ein Zeiger ist, und nicht direkt auf ein Objekt zeigt. Um eine Methode eines Objektes, in JavaScript-Notation zum Beispiel

```
GPUDevice.createBuffer(GPUBufferDescriptor)
```

, in C aufzurufen wird

```
wgpuDeviceCreateBuffer(WGPUDevice, const GPUBufferDescriptor*)
```

verwendet. Dadurch dass es in C keine Methoden gibt, sind die Funktionen alle global und haben das Prefix `wgpu`, danach kommt der Objekt-Typ (hier `Device`) und dann der eigentliche Name der Funktion (`CreateBuffer`). Zusätzlich muss als erster Parameter noch das Objekt (bzw. der Zeiger), auf das die Methode aufgerufen werden soll, übergeben werden.

4.1. Initialisierung

Die in diesem Abschnitt beschriebenen Schritte müssen normalerweise einmal am Start der Applikation getätigt werden, um das Rendering in Abschnitt **??** vorzubereiten. Im Vergleich zu „älteren“ Grafik-APIs wie OpenGL, DirectX pre 12 und insbesondere WebGL ist dies der größte Unterschied. Bei „modernen“ Grafik-APIs wird versucht, so viel Arbeit wie möglich beim Start zu erledigen, damit die eigentliche Rendering-Schleife so „schlank“ wie möglich sein kann.

4.1.1. GPU

Die Schnittstelle `GPU` markiert den Einstiegspunkt für **WebGPU**. Wenn **WebGPU** verfügbar ist, kann auf das `GPU`-Objekt in JavaScript über `navigator.gpu` zugegriffen werden.

```
1 var gpu = navigator.gpu;
```

4.1.2. GPUAdapter

Von dem `GPU`-Objekt lässt sich ein `GPUAdapter` anfordern, welcher die **WebGPU**-Implementierung auf dem jeweiligen System repräsentiert. Hierbei kann man optional noch Optionen zur Auswahl eines `GPUAdapter` angeben. Zum Beispiel, dass dedizierte GPUs integrierten bevorzugt werden sollen.

```
1 var adapter = await gpu.requestAdapter({powerPreference: "high-performance"});
```

4.1.3. GPUDevice

Hat man nun erfolgreich ein `GPUAdapter`-Objekt erhalten, kann man von diesem ein `GPUDevice` anfordern, welches die logische Repräsentation des Gerätes ist und zum Erstellen von Ressourcen auf der GPU benötigt wird. Hier kann man wiederum optional angeben, dass das gewünschte `GPUDevice` bestimmte *extensions* (zum Beispiel bestimmte Kompressionsverfahren) unterstützen oder bestimmte Mindestanforderungen bei den Ressourcen (zum Beispiel die unterstütze Mindestgröße für bestimmte *Buffer*) erfüllen soll.

```
1 var device = await adapter.requestDevice({extensions: ["texture-compression-bc"]});
```

4.1.4. GPUQueue

Eine `GPUQueue` erlaubt es, Befehle für die GPU zu sammeln und ihr zu übermitteln. Dabei werden die in der `GPUQueue` gesammelten Befehle der Reihe nach abgearbeitet, wenn diese abgeschickt wurde. Momentan gibt es nur eine `GPUQueue` pro `GPUDevice`, welche über dessen `defaultQueue`-Attribut erlangt werden kann.

```
1 var queue = device.defaultQueue;
```

Zu einem späteren Zeitpunkt, sollen allerdings mehrere `GPUQueue`s pro `GPUDevice` möglich sein [18].

4.1.5. `GPUBuffer`

Da dedizierte GPUs ihren eigenen Arbeitsspeicher, den sogenannten VRAM (Video-RAM), haben, kann die GPU nicht direkt mit den Daten im normalen Arbeitsspeicher (im folgenden RAM) arbeiten. Integrierte GPUs benutzen meistens den RAM als VRAM, jedoch wird er aus Anwendersicht zur Vereinheitlichung trotzdem als separat angesehen.

Um den VRAM der GPU nun also mit den benötigten Daten zu füllen, müssen dort sogenannte *Buffer* erstellt werden. Dies sind einfach Datenblöcke im VRAM mit einer gewissen Größe (in Bytes) und zusätzlich Informationen zu der beabsichtigten Verwendung der Daten. Die Datenblöcke kann man sich wie das Ergebnis des `malloc` Befehl in C vorstellen, der einen Datenblock mit einer gewissen Größe im RAM alloziert.

Um die Daten nun vom RAM in den VRAM zu bekommen, muss der *Buffer* in den RAM *gemapped* werden, also ein gleich großer Datenblock im RAM erstellt werden, der später in den VRAM kopiert wird. Dies kann nun entweder bei der Erstellung eines `GPUBuffer` (die **WebGPU**-Implementierung eines *Buffers*), gar nicht (wenn der *Buffer* nur von der GPU beschrieben und gelesen wird) oder erst im späteren Verlauf der Anwendung passieren.

In folgendem Beispiel erstellen wir einen `GPUBuffer` für unsere Vertexdaten. Um den Buffer später auch als *Vertex-Buffer* einsetzen zu können, muss die Option `GPUBufferUsage.VERTEX` gesetzt sein.

```
1 // Vertex Data (4 vertices with each 3 float for position and 3 float for color)
2 const vertex_data = new Float32Array([
3     -1.0, -1.0, 0.0,    1.0, 0.0, 0.0,
4      1.0, -1.0, 0.0,    0.0, 1.0, 0.0,
5     -1.0,  1.0, 0.0,    0.0, 0.0, 1.0,
6      1.0,  1.0, 0.0,    1.0, 1.0, 1.0,
7 ]);
8
9 var [vertex_buffer, buffer_data] = device.createBufferMapped({
10   size: vertex_data.byteLength,
11   usage: GPUBufferUsage.VERTEX
12 });
```

`createBufferMapped` liefert hier einerseits den erstellten `GPUBuffer` und anderseits direkt den *gemappeten* Speicherbereich als `ArrayBuffer`. In diesen können wir dann unsere Daten schreiben und den `GPUBuffer` *unmappen*, damit dieser von der GPU benutzt werden kann.

```
1 buffer_data = vertex_data;
2 vertex_buffer.unmap();
```

Dies war lange Zeit die einzige Möglichkeit einen `GPUBuffer` bei der Erstellung mit Daten zu initialisieren. Seit kurzem (zum Zeitpunkt des Verfassens), gibt es jedoch

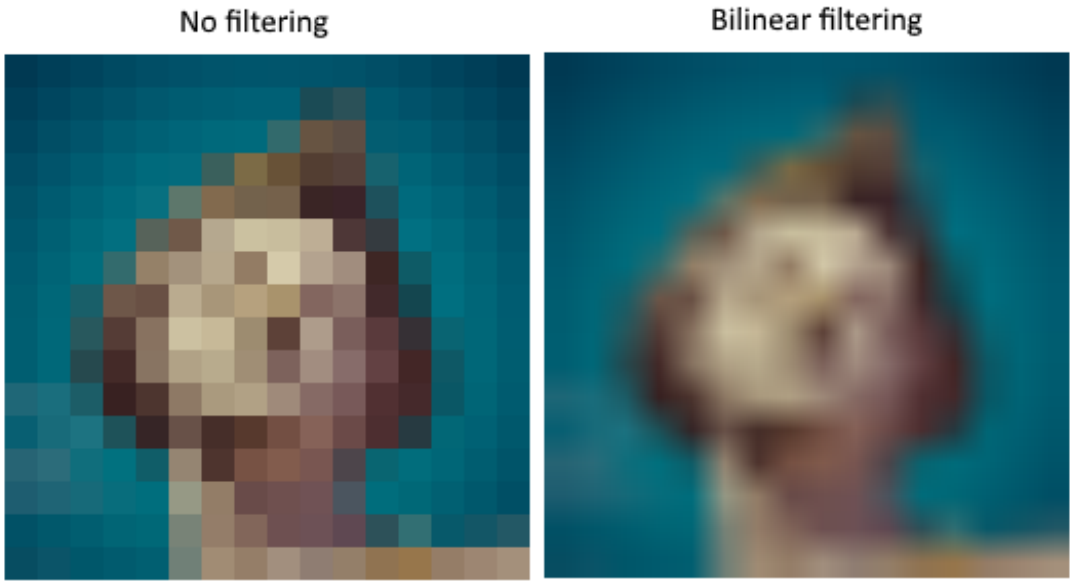

Abbildung 4.1.: Eine Textur als Lookup-Tabelle für Farbwerte. Ohne Interpolation (links) und mit bilinearer Interpolation (rechts). [41]

auch `writeBuffer` und `writeTexture` Methoden auf der `GPUQueue` [23], die diese Aufgaben übernehmen. Dies verwenden wir bei der Erstellung eines `GPUBuffer` für unsere Indexdaten:

```
1 // Index data (2 triangles with each 3 indexes)
2 const index_data = new Uint16Array([
3    0, 1, 2,
4    3, 2, 1
5 ]);
6
7 var index_buffer = device.createBuffer({
8   size: index_data.byteLength,
9   usage: GPUBufferUsage.INDEX
10 });
11 queue.writeBuffer(index_buffer, 0, index_data);
```

Dadurch, dass der `GPUBuffer` für `writeBuffer` nicht *gemapped* sein darf, verwenden wir hier `createBuffer` anstatt `createBufferMapped`. Dies liefert nur den *unmapped* `GPUBuffer` als Resultat zurück. Hier ist anzumerken, dass die `writeBuffer`-Methode zum Zeitpunkt des Verfassens dieses Abschnittes (28. Juni 2020) nur in Chromium, jedoch noch nicht in Firefox Nightly, implementiert ist.

4.1.6. GPUTexture

Neben Buffern sind Texturen die zweite wichtige Datenstruktur für die GPU. Eine Textur ist im Prinzip eine *Lookup-Tabelle* [51] mit 1-3 Dimensionen und der Möglichkeit zwischen den einzelnen *Zellen* zu interpolieren (siehe Abbildung 4.1).

Dabei sind für das Erstellen einer `GPUTexture` besonders die Eigenschaften `dimension`, `textureSize`, `format` und `textureUsage` wichtig. Die `dimension` gibt die Anzahl der Dimensionen (*1D*, *2D* oder *3D*) an. Die `textureSize` wiederum wie viele Texel (*texture element* [56]) es in den jeweiligen Dimension gibt. Das heißt, ein *1D*-Textur darf nur in einer Dimension mehr als ein Element haben (zum Beispiel (256, 1, 1)). Die einzelnen Eigenschaften der `textureSize` heißen dabei `width`, `height` und `depth`. Das `format` gibt an, was in jedem Texel gespeichert wird. Hier dürfte RGBA8 (also Rot, Grün, Blau und Alpha mit jeweils 8 Bit) wohl am bekanntesten sein. Jedoch gibt es sehr viele mehr, so sind aktuell 52 Texturformate in **WebGPU** verfügbar. `textureUsage` gibt an, wie die Texture benutzt werden soll, zum Beispiel zum Lesen oder zum Schreiben, als Kopierquelle oder Kopierziel, oder eine beliebige Kombination davon.

```
1 var texture = device.createTexture({
2   size: {
3     width: 1024,
4     height: 1024,
5     depth: 1
6   },
7   mipLevelCount: 1,
8   sampleCount: 1,
9   dimension: '2d',
10  format: 'rgba8unorm',
11  usage: GPUTextureUsage.SAMPLED | GPUTextureUsage.COPY_DST
12 });
```

Analog zu `writeBuffer` gibt es für Texturen auch eine `writeTexture`-Methode in der `GPUQueue`. Dies funktioniert allerdings nur, wenn die `GPUTexture` ein Kopierziel (`GPUTextureUsage.COPY_DST`) ist. Um Daten in die `GPUTexture` zu kopieren, muss dafür unter anderem erst eine `GPUTextureCopyView` erstellt werden, die zum Beispiel den Offset der Kopieroperation festlegt, so dass auch zum Beispiel Daten nur in die Mitte der Textur geschrieben werden können. Hier ist anzumerken, dass die `writeTexture`-Methode zum Zeitpunkt des Verfassens dieses Abschnittes (28. Juni 2020) weder in Chromium, noch in Firefox Nightly implementiert ist.

```
var texture_data = new Uint32Array(1024 * 1024);
texture_data.fill(0x00FF00FF); // 0 red, 255 blue, 0 green, 255 alpha

var texture_copy_view = {
    texture: texture,
    mipLevel: 0,
    origin: {
        x: 0,
        y: 0,
        z: 0
    }
};

var data_layout = {
    offset: 0,
    bytesPerRow: 1024 * 4,
    rowsPerImage: 1024
};

var copy_size = {
    width: 1024,
    height: 1024,
    depth: 1
};

queue.writeTexture(texture_copy_view, texture_data, data_layout, copy_size);
```

Um die `GPUTexture` nun wirklich nutzen zu können, wird noch eine `GPUTextureView` benötigt, die beschreibt wie genau auf die Daten der `GPUTexture` zugegriffen wird. In den meisten Fällen reicht allerdings die Erzeugung einer `GPUTextureView` von der `GPUTexture` ohne Angabe zusätzlicher Optionen. Dann übernimmt die `GPUTextureView` die Eigenschaften der `GPUTexture`:

```
var texture_view = texture.createView();
```

4.1.7. `GPUPipelineLayout`

Die `GPUPipelineBase` beschreibt die Basis für `GPURenderPipeline` und `GPUComputePipeline`. Die Gemeinsamkeit aller Pipelines besteht darin, dass sie unter anderem ein Set von Eingaben bekommen und daraus wiederum ein Set aus Ausgaben erstellen. Die Eingaben und Ausgaben bestehen dabei grundlegend aus Buffern und Texturen. Dabei muss beim Erstellen einer Pipeline ihr *Layout* festgelegt werden, d.h. was für Ressourcentypen benötigt werden und welche grundsätzlichen Eigenschaften diese Ressourcen haben. Die Eigenschaften umfassen zum Beispiel in welchem Shader (Vertex, Fragment oder Compute) die Ressourcen verfügbar sein sollen oder welche Dimension eine etwaige Textur hat.

Dabei ist zu beachten, dass hier nur die generelle Struktur der Ressourcen, aber nicht die Ressourcen an sich, festgelegt werden. Ein `GPUPipelineLayout` besteht dabei aus maximal 4 `GPUBindGroupLayout`s mit jeweils bis zu 16 `GPUBindGroupLayoutEntry`s. Diese Limitationen sind dabei die Mindestgrößen, die bei

WebGPU garantiert werden. Je nach Hardware könnte hier auch mehr verfügbar sein. Generell ist es aber zu empfehlen, die Mindestgrößen zu verwenden, um eine Funktionalität auf möglichst vielen Geräten zu ermöglichen.

Als *binding* wird die Verknüpfung einer Ressource mit dem jeweiligen Shader verstanden. Jedoch ist es nicht möglich ein einzelnes *binding* zu ändern, sondern immer nur eine *bind group*. Deshalb ist es sinnvoll die benötigten *bindings* so in die *bind groups* aufzuteilen, dass ein unnötiges Aktualisieren von *bindings*, die nicht aktualisiert werden müssen, vermieden wird. Als Beispiel könnte man hier einen Vertex-Shader sehen, der für jedes Modell die gleichen *View*- und *Projection*-Matrizen verwenden kann (da sich die Kamera nicht innerhalb eines *frames* verändert, aber jeweils die *Model*-Matrix aktualisieren muss (siehe Abschnitt 2.2.3). Hier könnte man dann zwei *bind groups* mit jeweils einem *binding* zu einem Buffer erstellen. Wobei ein Buffer hier die *View*- und *Projection*-Matrizen beinhaltet, so dass dieses *binding* nur einmal gesetzt werden muss. Das *binding* zum anderen Buffer, welcher die jeweilige *Model*-Matrix beinhaltet, muss dabei für jedes Modell aktualisiert werden.

```
1  // bgle = BindGroupLayoutEntry
2  // bgl = BindGroupLayout
3
4  var bgle_view_proj = {
5      binding: 0,
6      visibility: GPUShaderStage.VERTEX,
7      type: 'uniform-buffer',
8      hasDynamicOffset: false
9  };
10
11 var bgl_common = device.createBindGroupLayout({
12     entries: [
13         bgle_view_proj
14     ]
15 });
16
17 var bgle_model = {
18     binding: 0,
19     visibility: GPUShaderStage.VERTEX,
20     type: 'uniform-buffer',
21     hasDynamicOffset: true
22 };
23
24 var bgl_update = device.createBindGroupLayout({
25     entries: [
26         bgle_model
27     ]
28 });
29
30 var pipeline_layout = device.createPipelineLayout({
31     bindGroupLayouts: [
32         bgl_common,
33         bgl_update
34     ]
35 });
```

Das Attribut `binding` eines `BindGroupLayoutEntry` gibt hierbei einen Index an, der im Shader benutzt wird, um die Ressource innerhalb der *bind group* zu identifizieren. Im Shader wird die jeweilige *bind group* dann mit `set` identifiziert, was sich auf den Index des jeweiligen `GPUBindGroupLayout` in `GPUPipelineLayout.bindGroupLayouts` bezieht (siehe Listing 4.1).

```
1 layout(set = 0, binding = 0) uniform Common {
2   mat4 view;
3   mat4 proj;
4 } common;
5
6 layout(set = 1, binding = 0) uniform Update {
7   mat4 model;
8 } update;
9
10 ...
```

Listing 4.1: Deklaration der Buffer im Shader. `set` identifiziert hierbei die *bind group*, und `binding` die Ressource innerhalb der *bind group*.

4.1.8. `GPURenderPipeline`

Die `GPURenderPipeline` bildet die im Abschnitt 2.2 beschriebene Rendering-Pipeline im **WebGPU**-Kontext. Dabei produziert sie aus den Eingaben mithilfe von einer Reihe von jeweils festen oder frei programmierbaren Abschnitten Ausgaben. Die Eingaben bestehen dabei, zusätzlich zu den *bindings* jeder Pipeline, aus Vertex bzw. Indexdaten und *color* und *depth buffers*, die zum Start entweder geleert oder übernommen werden können. Die Ausgaben bestehen dabei aus den (eventuell) veränderten *color* und *depth buffers* und optional werden auch ein Teil der *bindings* (zum Beispiel *storage buffer*s oder *writeonly-storage-texture*s) verändert.

Die Rendering-Pipeline ist dabei in 6 nur konfigurierbare und 2 gänzlich frei programmierbare Abschnitte aufgeteilt (siehe Tabelle 4.1).

Mit diesen Informationen und dem `GPUPipelineLayout` aus Abschnitt 4.1.7 lässt sich nun eine `GPURenderPipeline` erstellen (siehe Listing 4.2).

```
1 var render_pipeline = device.createRenderPipeline({
2     layout: pipeline_layout,
3     vertexState: {
4         indexFormat: 'uint16',
5         vertexBuffers: [
6             {
7                 arrayStride: 24, // 24 bytes per vertex
8                 stepMode: 'vertex',
9                 attributes: [
10                    { // pos attribute
11                        format: 'float3',
12                        offset: 0,
13                        shaderLocation: 0
14                    },
15                    { // color attribute
```

#	WebGPU-Objekt	Beschreibung
1	`GPUVertexStateDescriptor`	Beschreibt das Format der Indexdaten (`Uint16` oder `Uint32`) und das Layout der einzelnen Vertexbuffer
2	`GPUShaderModule`	Vertex-Shader
3	`GPUPrimitiveTopology`	Beschreibt wie die Liste der Vertices aufgefasst werden soll (Punkte, Linien oder Dreiecke)
4	`GPURasterizationStateDescriptor`	Beschreibt wie die Vorderseite eines Dreiecks ermittelt werden soll und welche Seiten verworfen werden sollen
5	`GPUShaderModule`	Pixel- bzw. Fragment-Shader
6	`GPUDepthStencilStateDescriptor`	Beschreibt die Tests und Operationen für den *stencil* (Bitmaske) und *depth buffer*
7	`GPUColorStateDescriptor`	Beschreibt wie die Resultate von Abschnitt 5 in den *color buffer* gemischt werden sollen

Legende: nur konfigurierbar frei programmierbar

Tabelle 4.1.: Beschreibung der Abschnitte von `GPURenderPipeline`

```
16                          format: 'float3',
17                          offset: 12, // offset in bytes
18                          shaderLocation: 1
19                      }
20                  ]
21              }
22          ]
23      },
24      vertexStage: {
25          module: device.createShaderModule({
26              code: VertexShaderCode
27          }),
28          entryPoint: 'main'
29      },
30      primitiveTopology: 'triangle-list',
31      rasterizationState: {
32          frontFace = 'ccw', // counter clock wise
33          cullMode = 'none', // don't cull faces
34      },
35      fragmentStage: {
36          module: device.createShaderModule({
37              code: FragmentShaderCode
38          }),
39          entryPoint: 'main'
40      },
```

```
41      depthStencilState: {
42          format: 'depth32float',
43          depthWriteEnabled: true,
44          depthCompare: 'greater'
45      }
46      colorStates: [
47          {
48              format: 'rgba8srgb',
49              alphaBlend: {
50                  srcFactor: 'one',
51                  dstFactor: 'zero',
52                  operation: 'add'
53              },
54              colorBlend: {
55                  srcFactor: 'one',
56                  dstFactor: 'zero',
57                  operation: 'add'
58              },
59              writeMask: GPUColorWrite.ALL
60          }
61      ]
62  });
```

Listing 4.2: Erstellen einer „Standard“-`GPURenderPipeline`

4.1.9. `GPUBindGroup`

Um die Ressourcen nun in der `GPURenderPipeline` nutzen zu können, müssen diese in `GPUBindGroup`s zusammengefasst werden. Dabei beschreibt eine `GPUBindGroup` die tatsächliche Verknüpfung von Ressourcen auf Grundlage eines `GPUBindGroupLayout`. Analog dazu beschreibt ein `GPUBindGroupLayoutEntry` die grundlegende Struktur der zu verknüpfenden Ressource und `GPUBindGroupEntry` beschreibt die tatsächliche Verknüpfung. Dazu müssen wir aber erst die jeweiligen Ressourcen erstellen. In unserem Fall sind das 2 Buffer vom Typ *uniform*, d.h die Daten in den Buffern sind für jeden Shader-Aufruf gleich und unterscheiden sich nicht zwischen den einzelnen *vertices* (oder *fragments*). In einem Buffer werden dabei die *View*- und *Projection*-Matrizen gespeichert und in dem anderen die *Model*-Matrix (siehe Listing 4.3).

```
const fovy_rad = 60.0 * PI / 180.0;
const f = 1.0 / Math.tan(fovy_rad * 0.5);
const aspect = 800.0 / 600.0;
var common_data = new Float32Array([
    // view matrix
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0,
    // proj matrix
    f / aspect, 0.0, 0.0, 0.0,
    0.0, f, 0.0, 0.0,
    0.0, 0.0, 0.0, 0.1,
    0.0, 0.0, -1.0, 0.0
]);

var common_buffer = device.createBuffer({
    size: common_data.byteLength,
    usage: GPUBufferUsage.UNIFORM
});

queue.writeBuffer(common_buffer, 0, common_data);

var model_data = new Float32Array([
    // model matrix (moved 2 units in Z direction)
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 2.0,
    0.0, 0.0, 0.0, 1.0
]);

var dynamic_buffer = device.createBuffer({
    size: model_data.byteLength,
    usage: GPUBufferUsage.UNIFORM
});

queue.writeBuffer(dynamic_buffer, 0, model_data);
```

Listing 4.3: Erstellen der Uniform-Buffer für einerseits *common data* (*View-* / *Projection*-Matrix) und anderseits *dynamic data* (*Model*-Matrix).

Da wir in Abschnitt 4.1.7 2 `GPUBindGroupLayout`s mit jeweils 1 `GPUBindGroupLayoutEntry` erstellt haben, müssen wir nun passenden `GPUBindGroup`s und `GPUBindGroupEntry`s erstellen (siehe Listing 4.4).

```
var bind_group_common = device.createBindGroup({
    layout: bgl_common,
    entries: [
        {
            binding: 0,
            resource: {
                buffer: common_buffer,
                offset: 0,
                size: common_data.byteLength
            }
        }
    ]
});

var bind_group_dynamic = device.createBindGroup({
    layout: bgl_dynamic,
    entries: [
        {
            binding: 0,
            resource: {
                buffer: dynamic_buffer,
                offset: 0,
                size: model_data.byteLength
            }
        }
    ]
});
```

Listing 4.4: Erstellen der `GPUBindGroup`s für *common* und *dynamic data*.

4.1.10. `GPUSwapChain`

Um nun unsere Modelle zu rendern zu können brauchen wir noch eine Textur, in die wir das fertige Bild schreiben können, den *color buffer*. Diese Textur bekommen wir von der *swap chain* [54] zugeteilt. Dazu müssen wir aber erst ein `GPUSwapChain`-Objekt erstellen. Die `GPUSwapChain` können wir von einem `HTMLCanvasElement` bekommen (siehe Listing 4.5).

```
var canvas = document.getElementById('canvas');
var canvas_context = canvas.getContext('gpupresent');
var swap_chain = canvas_context.configureSwapChain({
    device: device,
    format: 'rgba8srgb',
    usage: GPUTextureUsage.OUTPUT_ATTACHMENT
});
```

Listing 4.5: Erstellen einer `GPUSwapChain` von einem `HTMLCanvasElement`

4.2. Rendering

Die Schritte in diesem Abschnitt müssen für jeden *frame* betätigt werden. Es ist zwar möglich, dies nur einmal zu machen, um zum Beispiel ein statisches Bild zu generieren. Für eine interaktive Applikation oder zum Beispiel einen Video-Player sollte die Wiederholrate an das Ausgabegerät angepasst werden, was meist um die 60 Hertz bzw. Bilder die Sekunde sind.

4.2.1. `GPUCommandEncoder`

Ein `GPUCommandEncoder` nimmt Befehle für die GPU auf. Um zum Beispiel den Inhalt eines `GPUBuffer` in einen anderen zu kopieren, kann die Methode `copyBufferToBuffer` benutzt werden. Hierbei ist aber wichtig zu beachten, dass die Befehle nicht direkt ausgeführt werden, sondern nur aufgenommen werden. Um die Befehle dann tatsächlich auf der GPU ausführen zu lassen, muss ein `GPUCommandBuffer` erstellt werden (siehe Abschnitt 4.2.3). Die Erstellung eines `GPUCommandEncoder` ist dabei trivial (siehe Listing 4.6).

```
1 var command_enc = device.createCommandEncoder();
```

Listing 4.6: Erstellen eines `GPUCommandEncoder`

4.2.2. `GPURenderPassEncoder`

Um nun die *Render*-Befehle aufzunehmen, muss erst ein `GPURenderPassEncoder` vom `GPUCommandEncoder erstellt werden`. Der `GPURenderPassEncoder` benötigt dabei zur Erstellung den *color* und *depth buffer*. Den *color buffer* bzw. die *color texture* erhalten wir dabei von der `GPUSwapChain` und den *depth buffer* bzw. die *depth texture* erstellen wir jedes Mal neu (siehe Listing 4.7).

```
var color_texture = swap_chain.getCurrentTexture();
var depth_texture = device.createTexture({
    size: {
        width: canvas.width,
        height: canvas.height,
        depth: 1
    },
    mipLevelCount: 1,
    sampleCount: 1,
    dimension: '2d',
    format: 'depth32float',
    usage: GPUTextureUsage.OUTPUT_ATTACHMENT
});

var render_pass = command_enc.beginRenderPass({
    colorAttachments: [
        {
            attachment: color_texture.createView(),
            loadValue: {r: 0.3, g: 0.0, b: 0.2, a: 1.0}, // clear to dark purple
            storeOp: 'store'
        }
    ],
    depthStencilAttachment: {
        attachment: depth_texture.createView(),
        depthLoadValue: 0.0,
        depthStoreOp: 'store',
        depthReadOnly: false,
        stencilLoadValue: 0,
        stencilStoreOp: 'store',
        stencilReadOnly: true
    },
});
```

Listing 4.7: Erstellen eines `GPURenderPassEncoder` vom `GPUCommandEncoder`. Die benötigte *color texture* bekommen wir von der `GPUSwapChain` und die benötigte *depth texture* erstellen wir jedes Mal neu.

Dem nun erstellten `GPURenderPassEncoder` können wir nun die `GPURenderPipeline`, die Vertex- und Index-Buffer, sowie die zwei *GPUBindGroup*s zuweisen und dann einen Befehl zum *rendern* des Modells aufnehmen (siehe Listing 4.8). Auch die Befehle zum Setzen der Ressourcen werden hier nur aufgenommen (in dem `GPUCommandEncoder` mit dem der `GPURenderPassEncoder` erstellt wurde) und erst später ausgeführt.

```
render_pass.setPipeline(render_pipeline);
render_pass.setVertexBuffer(0, vertex_buffer);
render_pass.setIndexBuffer(index_buffer);
render_pass.setBindGroup(0, bind_group_common);
render_pass.setBindGroup(1, bind_group_dynamic);
render_pass.drawIndexed(index_data.length);
render_pass.endPass();
```

Listing 4.8: Erstellen einer `GPUSwapChain` von einem `HTMLCanvasElement`

Abbildung 4.2.: Resultat des Minimalbeispiels in Google Chrome Canary.

4.2.3. `GPUCommandBuffer`

Haben wir nun alle benötigten Befehle aufgenommen können wir ein `GPUCommandBuffer` vom `GPUCommandEncoder` generieren und die Befehle mithilfe der `GPUQueue` zur GPU schicken (siehe Listing 4.9).

```
1 var command_buffer = command_enc.finish();
2 queue.submit([command_buffer]);
```

Listing 4.9: Generieren eines `GPUCommandBuffer` und Abschicken der Befehle an die GPU.

4.3. Resultat

Wie in Abbildung 4.2 zu sehen, erhalten wir nun ein farbiges Bild in unserem HTML-Canvas. Dabei haben wir nur die Eckpunkte und ihre jeweiligen Farben spezifiziert und die GPU hat die Farben für die inneren Pixel aus den Vertexdaten interpoliert.

Dabei ist zu beachten, dass dieses Minimalbeispiel nicht darauf ausgelegt ist ein beachtliches Resultat zu erzeugen, sondern es soll mehr einen Querschnitt durch die **WebGPU** aufzeigen und veranschaulichen, was dazu nötig ist, eine sehr simple Applikation zu erstellen. Da der größte Teil des Aufwands hierbei in der Initialisierung steckt, kann die Applikation nun mit relativ wenig Aufwand erweitert

werden. Dazu müssen primär nur noch die einzelnen Ressourcen (Vertex-, Index- und Uniform-Buffer) und die Shader erweitert werden. An der Grundstruktur muss nicht mehr viel angepasst werden.

5. Die spider-Engine

Für das in diesem Kapitel beschriebene Projekt wurde Windows 10 mit Ubuntu 19.10 per Windows Subsystem for Linux (WSL) [30] verwendet. Dabei wurden die Quellcodedateien 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.

5.1. 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.*

5.1.1. Überblick

Mit der parallel zu dieser Arbeit entstandenen **spider**-Engine (`https://github.com/Tandaradei/webgpu-wasm`), lässt sich mit geringem Aufwand eine Webapplikation zur Darstellung von 3D-Szenen in C/C++ erstellen. Das Grundgerüst dafür ist sehr minimal (siehe Listing 5.1).

```
#include "spider/spider.h"

// Create the initial state of your scene
void initApplication();
// Update your scene each frame
bool update(float delta_time_s);

int main() {
  const uint32_t surface_width = 1280;
  const uint32_t surface_height = 720;

  // Initialize the spider engine
  spInit(&(SPInit){
    .surface_size = {
      .width = surface_width,
      .height = surface_height
    },
    .update_func = update,
    .camera = {
      .pos = {0.0f, 2.0f, 5.0f},
      .look_at = {0.0f, 0.0f, 0.0f},
      .mode = SPCameraMode_LookAt,
      .fovy = glm_rad(60.0f),
      .aspect = (float)surface_width / (float) surface_height,
      .near = 0.1f,
    },
  });

  initApplication();
  spStart();

  return 0;
}

void initApplication() {
  // Here you can create lights (only spotlights for now),
  // create custom meshes and materials
  // or load them from glTF files (recommended)
}

bool update(float delta_time_s) {
  // Here you can update your created lights, objects and the main camera

  // Return false, if you want to quit the application
  return true;
}
```

Listing 5.1: 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 5.1 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.

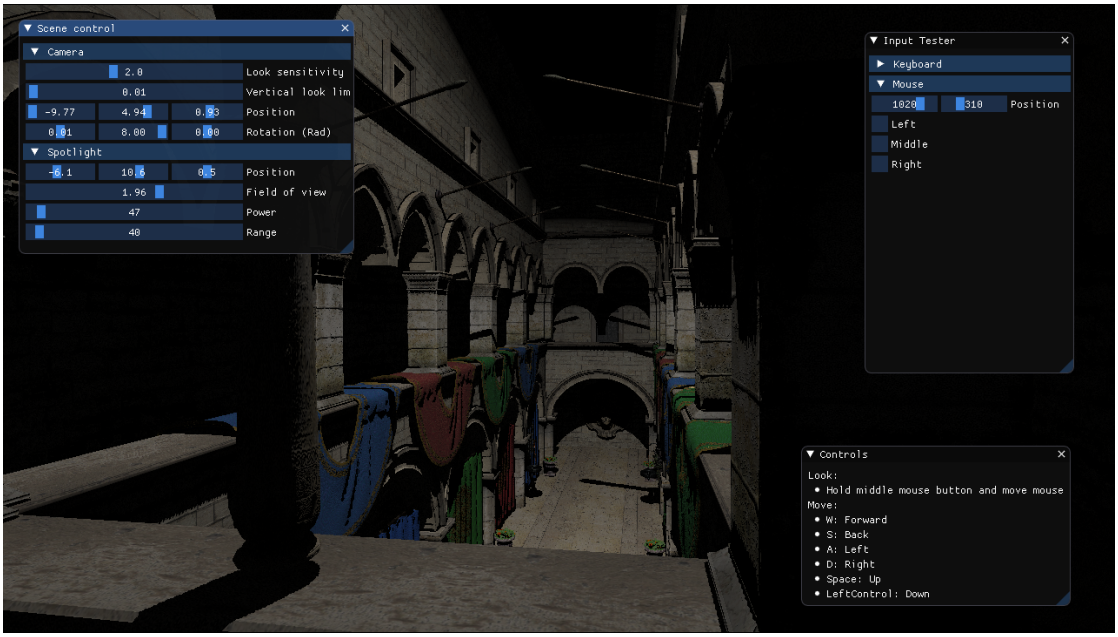

Abbildung 5.1.: Interaktive 3D-Szene mit UserInterface. Bildschirmaufnahme durch Verfasser.

5.1.2. 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 anderseits 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`
Preprozessordirektive	`SP_`	`SP_INVALID_ID`
globale Variable	`sp_`	*kein Beispiel*
„privat“		
Struktur	`_SP`	`_SPRenderPipeline`
Enum	`_SP`	*kein Beispiel*
Funktion	`_sp`	`_spSetupPools`
Preprozessordirektive	`_SP_`	`_SP_MATERIAL_POOL_DEFAULT`
globale Variable	`_sp_`	`_sp_state`

Tabelle 5.1.: In der **spider**-Engine verwendete Prefixe

Prefixe

Da es in C keine Namensbereiche, wie zum Beispiel in C++, gibt, werden alle Strukturen, Enums, Funktionen, Preprozessordirektiven und globale Variablen mit einem Prefix versehen (siehe Tabelle 5.1). 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 5.2 definiert.

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

Listing 5.2: 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. 44].

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 5.3).

```
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 5.3: Beispiele zu bestimmten Initialisierern (Unter Verwendung der in Listing 5.2 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 5.4).

```
typedef struct MyFunctionDesc {
  uint32_t values[5];
  const char* name;
} MyFunctionDesc;

void myFunction(const MyFunctionDesc* desc);

MyFunctionDesc my_desc;
my_desc.values[2] = 5;
my_desc.name = "Abc";
myFunction(&my_desc);

myFunction(&(MyFunctionDesc){
  .values = {
    [2] = 5,
  },
  .name = "Abc",
});
```

Listing 5.4: Verwendung einer *Desc*-Struktur zum Übergeben von Argumenten an eine Funktion

handles (Vgl. 43)

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 [43] 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 5.5). Dieser Zeiger sollte vom Benutzter 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 5.5: Verwendung von *handles*

5.2. 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**-Application Programming Interface (dt. Programmierschnittstelle) (API) benutzen soll, muss der C-Quellcode in ein Format umgewandelt werden, welches ein Webbrowser verstehen kann. Dazu wird das Werkzeug **Emscripten** verwendet. Mit **Emscripten** lässt sich (unter anderem) C-Quellcode mit Hilfe von LLVM [25] zu JavaScript und WebAssembly [42] kompilieren.

5.2.1. 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 **Emscripten** 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-Quellcodedatei kompiliert werden:

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

Dadurch entstehen folgende Dateien:

- `test.html`: Eintrittspunkt für die Web-Applikation
- `test.wasm`: Kompilierter C-Code in WebAssembly
- `test.js`: JavaScript-Code zur Verknüfung von HTML, JavaScript und WebAssembly

5.3. 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* [20] und *NVIDIA NSight Graphics* [35] Probleme damit, mit den benutzten Webbrowser-Versionen *Chrome Canary* [7] und *Firefox Nightly* [31] 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* [27] (eine Softwareschicht um *Direct3D 11*-Befehle auf *Direct3D 12*-Befehlen zu übersetzen), das in *Chrome* benutzt wird, gar nicht unterstüzt wird. Einzig mit *PIX on Windows* [29] 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 5.2 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 5.3).

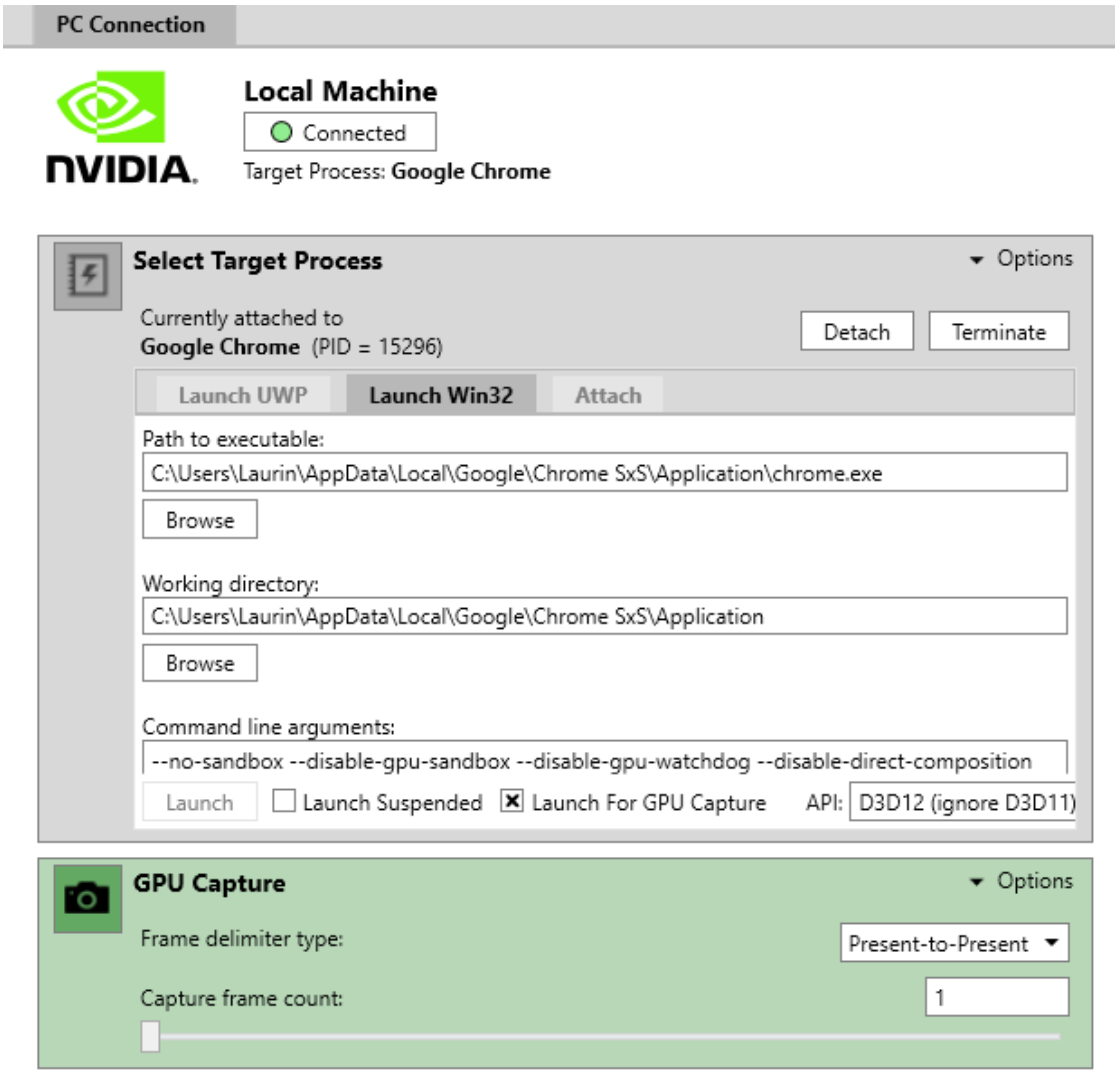

Abbildung 5.2.: Übersicht der Optionen zum Starten von *Google Chrome Canary* in *Microsoft PIX on Windows* [29]. Bildschirmaufnahme durch Verfasser.

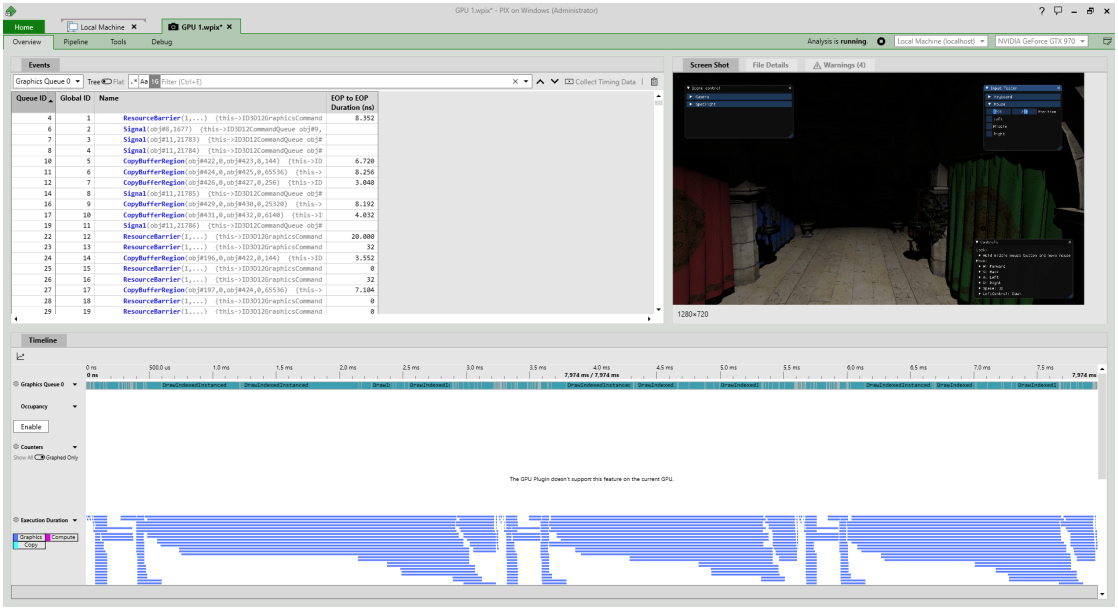

Abbildung 5.3.: Anzeige nach der Erfassung eines Einzelbildes in *Microsoft PIX on Windows* [29]. Bildschirmaufnahme durch Verfasser.

6. Zusammenfassung und Ausblick

6.1. Erreichte Ergebnisse

Durch die Arbeit hat der Verfasser ein tieferes Verständnis von **WebGPU** (und anderen Grafik-APIs) erhalten. Dadurch konnte unter anderem die **spider**-Engine erfolgreich als praktischer Teil umgesetzt werden, so dass andere diese benutzen können um ihre eigenen interaktiven 3D-Web-Applikationen zu erstellen.

Auch ist ein besseres Verständnis für das Compiler-Werkzeug emscripten zustande gekommen. Dies ging sogar so weit, dass der Verfasser, die **WebGPU**-Integration in emscripten aktualisieren und verbessern konnte (siehe `https://github.com/emscripten-core/emscripten/pull/11067`).

6.2. Ausblick

Sobald die **WebGPU**-Spezifikation und Implementierung vollständig (Version 1.0) ist, lassen sich die Ergebnisse dieser Arbeit neu evaluieren und man kann dann den Fortschritt daran messen. Da die in dieser Arbeit beschriebenen Teile von **WebGPU** schon relativ fest stehen und sich zur „fertigen“ Version wahrscheinlich nicht mehr großartig ändern werden, könnte sich eine weitere Arbeit explizit mit den (dann hoffentlich besser spezifizierten und implementierten) *GPU-Compute*-Funktionen beschäftigen.

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 28. 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 28. 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 28. 06. 2020).

[7] Google Corporation. *Chrome Canary*. Software (Anwendung). URL: https://www.google.com/intl/de/chrome/canary/ (besucht am 28. 06. 2020).

[8] Google Corporation. *Chromium*. Software (Anwendung). URL: https://www.chromium.org/Home (besucht am 28. 06. 2020).

[9] Google Corporation. *Dawn, a WebGPU implementation*. Software (Implementierung). URL: https://dawn.googlesource.com/dawn (besucht am 28. 06. 2020).

[10] GPU for the Web Community Group. *GPU for the Web Community Group*. URL: https://www.w3.org/community/gpu/ (besucht am 28. 06. 2020).

[11] GPU for the Web Community Group. *Implementation Status*. URL: https://github.com/gpuweb/gpuweb/wiki/Implementation-Status (besucht am 28. 06. 2020).

[12] GPU for the Web Community Group. *WebGPU*. Hrsg. von Dzmitry Malyshau, Justin Fan und Kai Ninomiya. Editor's Draft, 8 June 2020. URL: https://gpuweb.github.io/gpuweb/ (besucht am 28. 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 28. 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 28. 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 28. 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 28. 06. 2020).

[17] Joshua-Ashton. *WGSL is terrible!* Online-Diskussion. URL: https://github.com/gpuweb/gpuweb/issues/566 (besucht am 28. 06. 2020).

[18] kainino0x. *Change .getQueue() to .defaultQueue*. Online-Diskussion. URL: https://github.com/gpuweb/gpuweb/pull/490 (besucht am 28. 06. 2020).

[19] Kangz. *Consider adding sugar for familiar loop constructs*. Online-Diskussion. URL: https://github.com/gpuweb/gpuweb/issues/569 (besucht am 28. 06. 2020).

[20] Baldur Karlsson und RenderDoc community. *RenderDoc*. Software (Anwendung). URL: https://renderdoc.org (besucht am 28. 06. 2020).

[21] Khronos Group. *glTF Overview*. Datenformat. URL: https://www.khronos.org/gltf/ (besucht am 28. 06. 2020).

[22] 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 28. 06. 2020).

[23] kvark. *Add Queue/writeBuffer method*. Online-Diskussion. URL: https://github.com/gpuweb/gpuweb/pull/749 (besucht am 28. 06. 2020).

[24] learnopengl.com. *Overview of coordinate systems*. Bild. URL: https://learnopengl.com/Getting-started/Coordinate-Systems (besucht am 28. 06. 2020).

[25] LLVM community. *LLVM*. Software (Werkzeug). URL: https://llvm.org/ (besucht am 28. 06. 2020).

[26] 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.

[27] Microsoft Corporation. *D3D11On12*. Software (API). URL: https://github.com/microsoft/D3D11On12 (besucht am 28. 06. 2020).

[28] Microsoft Corporation. *Office 365*. URL: https://www.office.com/?ms.officeurl=webapps (besucht am 28. 06. 2020).

[29] Microsoft Corporation. *PIX on Windows*. Software (Anwendung). URL: https://devblogs.microsoft.com/pix/ (besucht am 28. 06. 2020).

[30] Microsoft Corporation. *What is the Windows Subsystem for Linux?* Software (Werkzeug). URL: https://docs.microsoft.com/de-de/windows/wsl/about (besucht am 28. 06. 2020).

[31] Mozilla Foundation. *Firefox Nighlty*. Software (Anwendung). URL: https://www.mozilla.org/de/firefox/all/#product-desktop-nightly (besucht am 28. 06. 2020).

[32] Mozilla Foundation. *wgpu*. Software (Implementierung). URL: https://hg.mozilla.org/mozilla-central/file/tip/gfx/wgpu (besucht am 28. 06. 2020).

[33] NVIDIA Corporation. *NVIDIA GeForce GTX 1650*. Hardware. URL: https://www.nvidia.com/de-de/geforce/graphics-cards/gtx-1650/ (besucht am 28. 06. 2020).

[34] NVIDIA Corporation. *NVIDIA GeForce RTX 2080Ti*. Hardware. URL: https://www.nvidia.com/de-de/geforce/graphics-cards/rtx-2080-ti (besucht am 28. 06. 2020).

[35] NVIDIA Corporation. *NVIDIA® Nsight™ Graphics*. Software (Anwendung). URL: https://developer.nvidia.com/nsight-graphics (besucht am 28. 06. 2020).

[36] Online users. *Discussion on: WebGPU Shading Language*. Online-Diskussion. URL: https://news.ycombinator.com/item?id=22536961 (besucht am 28. 06. 2020).

[37] Pixar. *RenderMan Shading Language*. Spezifikation. URL: https://renderman.pixar.com/resources/RenderMan_20/shadingLanguage.html (besucht am 28. 06. 2020).

[38] 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 28. 06. 2020).

[39] shaderc Contributors. *shaderc*. Software (Werkzeug). URL: https://github.com/google/shaderc (besucht am 28. 06. 2020).

[40] StatCounter. *Browser Market Share Worldwide - May 2020*. URL: https://gs.statcounter.com/browser-market-share (besucht am 28. 06. 2020).

[41] Vulkan Tutorial. *Texture filtering*. Bild. URL: https://vulkan-tutorial.com/images/texture_filtering.png (besucht am 28. 06. 2020).

[42] WebAssembly community. *WebAssembly*. Spezifikation. URL: https://webassembly.org/ (besucht am 28. 06. 2020).

[43] Andre Weissflog. *Handles are the better pointers*. URL: https://floooh.github.io/2018/06/17/handles-vs-pointers.html (besucht am 28. 06. 2020).

[44] Andre Weissflog. *Modern C for C++ Peeps*. URL: https://floooh.github.io/2019/09/27/modern-c-for-cpp-peeps.html (besucht am 28. 06. 2020).

[45] 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 28. 06. 2020).

[46] Wikipedia, the free encyclopedia. *3dfx Voodoo Graphics*. URL: https://de.wikipedia.org/wiki/3dfx_Voodoo_Graphics (besucht am 28. 06. 2020).

[47] Wikipedia, the free encyclopedia. *GeForce 256*. URL: https://en.wikipedia.org/wiki/GeForce_256 (besucht am 28. 06. 2020).

[48] Wikipedia, the free encyclopedia. *GeForce 3 series*. URL: https://en.wikipedia.org/wiki/GeForce_3_series (besucht am 28. 06. 2020).

[49] Wikipedia, the free encyclopedia. *Google Chrome*. URL: https://de.wikipedia.org/wiki/Google_Chrome (besucht am 28. 06. 2020).

[50] Wikipedia, the free encyclopedia. *High-Level Shading Language*. URL: https://en.wikipedia.org/wiki/High-Level_Shading_Language (besucht am 28. 06. 2020).

[51] Wikipedia, the free encyclopedia. *Lookup-Tabelle*. URL: https://de.wikipedia.org/wiki/Lookup-Tabelle (besucht am 28. 06. 2020).

[52] Wikipedia, the free encyclopedia. *Opera (Browser)*. URL: https://de.wikipedia.org/wiki/Opera_(Browser) (besucht am 28. 06. 2020).

[53] Wikipedia, the free encyclopedia. *Samsung Internet*. URL: https://en.wikipedia.org/wiki/Samsung_Internet (besucht am 28. 06. 2020).

[54] Wikipedia, the free encyclopedia. *Swap chain*. URL: https://en.wikipedia.org/wiki/Swap_chain (besucht am 28. 06. 2020).

[55] Wikipedia, the free encyclopedia. *Swizzling (computer graphics)*. URL: https://en.wikipedia.org/wiki/Swizzling_(computer_graphics) (besucht am 28. 06. 2020).

[56] Wikipedia, the free encyclopedia. *Texel (graphics)*. URL: https://en.wikipedia.org/wiki/Texel_(graphics) (besucht am 28. 06. 2020).

[57] Wikipedia, the free encyclopedia. *WebGPU*. URL: https://en.wikipedia.org/wiki/WebGPU (besucht am 28. 06. 2020).

A. Anhang A

Dies ist der Quellcode zu der in Abbildung 5.1 gezeigten Beispielanwendung. Die Anwendung lädt dabei das relativ komplexe 3D-Modell **Sponza** [22] aus einer glTF-Datei [21], 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
           'incompatible-pointer-types-discards-qualifiers'
19     // cglm takes no const pointers as arguments, even if it doesn't mutate the
           vectors
20     glm_vec3_sub((float*)light_look_at, (float*)light_pos, light_direction);
21     glm_vec3_normalize(light_direction);
22
23     spot_light_id = spCreateSpotLight(&(SPSpotLightDesc){
24             .pos = {light_pos[0], light_pos[1], light_pos[2]},
25             .range = 40.0f,
26             .color = {.r = 255, .g = 255, .b = 255},
27             .dir = {light_direction[0], light_direction[1], light_direction[2]},
28             .fov = glm_rad(70.0f),
29             .power = 20.0f,
30             .shadow_casting = &(SPLightShadowCastDesc){
31                 .shadow_map_size = 2048,
32             },
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 *
                    0.5f, "%.2f", 1.0f);
96              igSliderFloat3("Position##cam", (float*)&cam->pos, -10.0f, 10.0f,
                    "%.2f", 1.0f);
97              igSliderFloat3("Rotation (Rad)##cam", (float*)&cam_rot, -M_PI, M_PI,
                    "%.2f", 1.0f);
98              igSliderFloat("Vertical field of view (Rad)##cam", &cam->fovy, 0.01f,
                    M_PI, "%.2f", 1.0f);
99          }
100     }
101
102     last_mouse_pos_x = spGetMousePositionX();
103     last_mouse_pos_y = spGetMousePositionY();
104
105     SPLight* spot_light = spGetLight(spot_light_id);
106     if(spot_light) {
107         if(igCollapsingHeaderTreeNodeFlags("Spotlight", ImGuiTreeNodeFlags_None)) {
108             igSliderFloat3("Position##light", (float*)&spot_light->pos, -50.0f,
                    50.0f, "%.1f", 1.0f);
109             igSliderFloat("Field of view##light", &spot_light->fov, 0.0f, M_PI,
                    "%.2f", 1.0f);
110             igSliderFloat("Power##light", &spot_light->power, 0.0f, 1000.0f,
                    "%.0f", 1.0f);
111             igSliderFloat("Range##light", &spot_light->range, 0.0f, 1000.0f,
                    "%.0f", 1.0f);
112         }
113     }
114     igEnd();
115
116     // return false if you want to quit
117     return true;
118 }
119
120 int main() {
121     spInit(&(SPInitDesc){
122         .surface_size = {
123             .width = surface_width,
124             .height = surface_height
125         },
126         .update_func = update,
127         .camera = {
128             .pos = {0.0f, 2.0f, 0.0f},
129             .dir = {0.0f, 0.0f, 1.0f},
130             .look_at = {0.0f, 0.0f, 0.0f},
131             .mode = SPCameraMode_Direction,
132             .fovy = glm_rad(60.0f),
133             .aspect = (float)surface_width / (float) surface_height,
134             .near = 0.1f,
135         },
136         .pools.capacities = {
137             .meshes = 128,
138             .materials = 64,
139             .render_meshes = 256,
140             .lights = 1,
141             .scene_nodes = 1024,
142         },
143         .show_stats = true,
144     });
145
146     init();
147
148     spStart();
```

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

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

B. Anhang B

```
#version 450
#extension GL_ARB_separate_shader_objects : enable
#extension  GL_EXT_samplerless_texture_functions : enable

layout(set = 0, binding = 1) uniform Camera {
    mat4 view;
    mat4 proj;
    vec3 pos;
} cam;

layout(set = 0, binding = 2) uniform Light {
    mat4 view;
    mat4 proj;
    vec4 pos3_range1;
    vec4 color3_type1; // type: 0 directional, 1 spot, 2 point
    vec4 dir3_fov1; // dir: for spot & dir, fov: for spot
    vec4 area2_power1_padding1; // area: for dir
} light;  // TODO: support more than 1 light

layout(set = 2, binding = 0) uniform texture2D albedo_tex;
layout(set = 2, binding = 1) uniform sampler albedo_sampler;
layout(set = 2, binding = 2) uniform texture2D normal_tex;
layout(set = 2, binding = 3) uniform sampler normal_sampler;
layout(set = 2, binding = 4) uniform texture2D ao_roughness_metallic_tex;
layout(set = 2, binding = 5) uniform sampler arm_sampler;
layout(set = 2, binding = 6) uniform texture2D shadow_map;
layout(set = 2, binding = 7) uniform sampler shadow_sampler;


layout(location = 0) in vec3 fragPosWorld;
layout(location = 1) in vec2 fragTexCoords;
layout(location = 2) in vec3 fragNormal;
layout(location = 3) in vec3 fragTangent;

layout(location = 0) out vec4 outColor;

const float PI = 3.14159265359;
const float gamma = 2.2;

const int sampleSize = 1;
const float scale = 1.5;

float getLightDepthOnPosSingle(vec2 coords_shadow_map) {
    return texture( sampler2D( shadow_map, shadow_sampler ), coords_shadow_map.xy
        ).r;
}

float getLightDepthOnPosSampled(vec2 coords_shadow_map) {
    float value = 0.0;
    ivec2 texDim = textureSize(shadow_map, 0);
    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 *
                    dy));
56              count++;
57          }
58      }
59      return value / count;
60  }
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),
            fragTexCoords).rgba;
99      const vec3 albedo = albedo_all.rgb;
100     const float alpha = albedo_all.a;
101     if(alpha == 0.0) {
102         discard;
103     }
104     vec3 tangent_space_normal = normalize(texture(sampler2D(normal_tex,
            normal_sampler), fragTexCoords).xyz * vec3(2.0, 2.0, 1.0) - vec3(1.0, 1.0,
            0.0));
105     const vec3 ao_roughness_metallic = texture(sampler2D(ao_roughness_metallic_tex,
            arm_sampler), fragTexCoords).rgb;
106     const float ao          = ao_roughness_metallic.r;
107     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) /
                3.14, 1.0 - (light_fov * light_fov));
136     }
137     // Shadow calculation
138     const vec4 pos_shadow_map = light.proj * light.view * vec4(fragPosWorld, 1.0);
139     vec4 pos_in_light_clip_space = pos_shadow_map / pos_shadow_map.w;
140     pos_in_light_clip_space.xy = pos_in_light_clip_space.xy * 0.5 + 0.5; // [-1,1]
            to [0,1]
141     pos_in_light_clip_space.y = 1.0 - pos_in_light_clip_space.y; // bottom-up to
            top-down
142     // [0, 0] of pos_in_light_clip_space.xy should now be the top left corner and
            [1, 1] the bottom right --> texture space
143
144     const float depth_bias = 0.00001; // Depth bias is not yet implemented in
            Chromium/dawn, so we have to "fake" it in the shader
145     if(attenuation > 0.0 && pos_in_light_clip_space.z > 0.0 &&
            pos_in_light_clip_space.z < 1.0) {
146         const float light_depth_on_pos =
                getLightDepthOnPosSampled(pos_in_light_clip_space.xy);
147         if(pos_in_light_clip_space.w > 0.0 && light_depth_on_pos - depth_bias >
                pos_in_light_clip_space.z) {
148             attenuation = 0.0;
149         }
150     }
151     if(attenuation > 0.0) {
152         // calculate per-light radiance
153         vec3 H = normalize(V + L);
154         vec3 radiance     = light.color3_type1.rgb * attenuation;
155
156         // cook-torrance brdf
157         float NDF = DistributionGGX(N, H, roughness);
158         float G   = GeometrySmith(N, V, L, roughness);
159         vec3 F    = fresnelSchlick(max(dot(H, V), 0.0), F0);
160
161         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

C. Anhang C

```
#version 450

layout(set = 0, binding = 0) uniform Common {
    mat4 view;
    mat4 proj;
};

layout(set = 1, binding = 0) uniform Dynamic {
    mat4 model;
};

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inColor;

layout (location = 0) out vec3 outColor;

void main() {
    vec4 pos = proj * view * model * vec4(inPos, 1.0);
    outColor = inColor;
    gl_Position = pos;
}
```

Listing C.1: Kompletter GLSL-Code des Vertex-Shaders für das Minimalbeispiel aus Kapitel 4

```
#version 450

layout (location = 0) in vec3 inColor;

layout (location = 0) out vec4 outFragColor;

void main() {
    outFragColor = vec4(inColor, 1.0);
}
```

Listing C.2: Kompletter GLSL-Code des Fragment-Shaders für das Minimalbeispiel aus Kapitel 4

```
<html>
<body>
    <canvas id="canvas" width="800" height="600"></canvas>
    <script src="webgpu.js"></script>
</body>
</html>
```

Listing C.3: Kompletter HTML-Quellcode für das Minimalbeispiel aus Kapitel 4

```
async function loadShader(shaderPath) {
```

```
2     return await fetch(new Request(shaderPath), { method: 'GET', mode: 'cors'
          }).then((res) =>
3         res.arrayBuffer().then((arr) => new Uint32Array(arr))
4     );
5 }
6
7 function initBuffer(data, usage, device, queue) {
8     const use_write_buffer = true;
9     if(use_write_buffer && queue["writeBuffer"] !== undefined){
10        var buffer = device.createBuffer({
11            size: data.byteLength,
12            usage: usage | GPUBufferUsage.COPY_DST
13        });
14        queue.writeBuffer(buffer, 0, data.buffer);
15        return buffer;
16    }
17    else {
18        var [buffer, buffer_data] = device.createBufferMapped({
19            size: data.byteLength,
20            usage: usage
21        });
22        var writeArray = data instanceof Uint16Array ? new Uint16Array(buffer_data)
              : new Float32Array(buffer_data);
23        writeArray.set(data);
24        buffer.unmap();
25        return buffer;
26    }
27 }
28
29 async function start() {
30    var gpu = navigator.gpu;
31    var adapter = await gpu.requestAdapter({powerPreference: "high-performance"});
32    var device = await adapter.requestDevice();
33    var queue = device.defaultQueue;
34    // Vertex Data (4 vertices with each 3 float for position and 3 float for color)
35    const vertex_data = new Float32Array([
36    //     x,    y,   z,      r,   g,   b
37        -1.0,  1.0, 0.5,    1.0, 0.0, 0.0,
38         1.0,  1.0, 0.5,    0.0, 1.0, 0.0,
39        -1.0, -1.0, 0.5,    0.0, 0.0, 1.0,
40         1.0, -1.0, 0.5,    1.0, 1.0, 1.0,
41    ]);
42
43    var vertex_buffer = initBuffer(vertex_data, GPUBufferUsage.VERTEX, device,
          queue);
44
45    // Index data (2 triangles with each 3 indices)
46    const index_data = new Uint16Array([
47        0, 1, 2,
48        2, 1, 3
49    ]);
50
51    var index_buffer = initBuffer(index_data, GPUBufferUsage.INDEX, device, queue);
52
53    var texture = device.createTexture({
54        size: {
55            width: 1024,
56            height: 1024,
57            depth: 1
58        },
59        mipLevelCount: 1,
60        sampleCount: 1,
```

```
61          dimension: '2d',
62          format: 'rgba8unorm',
63          usage: GPUTextureUsage.SAMPLED | GPUTextureUsage.COPY_DST
64      });
65      console.log(texture);
66
67      var texture_data = new Uint32Array(1024 * 1024);
68      texture_data.fill(0x00FF00FF); // 0 red, 255 blue, 0 green, 255 alpha
69
70      var texture_copy_view = {
71          texture: texture,
72          mipLevel: 0,
73          origin: {
74              x: 0,
75              y: 0,
76              z: 0
77          }
78      };
79
80      var data_layout = {
81          offset: 0,
82          bytesPerRow: 1024 * 4,
83          rowsPerImage: 1024
84      };
85
86      var copy_size = {
87          width: 0,
88          height: 0,
89          depth: 0
90      };
91      if(queue["writeTexture"] !== undefined) {
92          queue.writeTexture(texture_copy_view, texture_data, data_layout, copy_size);
93      }
94      else {
95          console.log("writeTexture not implemented");
96      }
97
98      // bgle = BindGroupLayoutEntry
99      // bgl = BindGroupLayout
100
101     var bgle_view_proj = {
102         binding: 0,
103         visibility: GPUShaderStage.VERTEX,
104         type: 'uniform-buffer',
105         hasDynamicOffset: false
106     };
107
108     var bgl_common = device.createBindGroupLayout({
109         entries: [
110             bgle_view_proj
111         ]
112     });
113
114     var bgle_model = {
115         binding: 0,
116         visibility: GPUShaderStage.VERTEX,
117         type: 'uniform-buffer',
118         hasDynamicOffset: false
119     };
120
121     var bgl_dynamic = device.createBindGroupLayout({
122         entries: [
```

```
123             bgle_model
124         ]
125     });
126
127     var pipeline_layout = device.createPipelineLayout({
128         bindGroupLayouts: [
129             bgl_common,
130             bgl_dynamic
131         ]
132     });
133
134     var canvas = document.getElementById('canvas');
135     var canvas_context = canvas.getContext('gpupresent');
136     const canvas_format = await canvas_context.getSwapChainPreferredFormat(device);
137     var swap_chain = canvas_context.configureSwapChain({
138         device: device,
139         format: canvas_format,
140         usage: GPUTextureUsage.OUTPUT_ATTACHMENT
141     });
142
143     const vertex_shader_code_spirv = await loadShader('simple.vert.spv');
144     const frag_shader_code_spirv = await loadShader('simple.frag.spv');
145
146     var render_pipeline = device.createRenderPipeline({
147         layout: pipeline_layout,
148         vertexState: {
149             indexFormat: 'uint16',
150             vertexBuffers: [
151                 {
152                     arrayStride: 24, // 24 bytes per vertex
153                     stepMode: 'vertex',
154                     attributes: [
155                         { // pos attribute
156                             format: 'float3',
157                             offset: 0,
158                             shaderLocation: 0
159                         },
160                         { // color attribute
161                             format: 'float3',
162                             offset: 12, // offset in bytes
163                             shaderLocation: 1
164                         }
165                     ]
166                 }
167             ]
168         },
169         vertexStage: {
170             module: device.createShaderModule({
171                 code: vertex_shader_code_spirv
172             }),
173             entryPoint: 'main'
174         },
175         primitiveTopology: 'triangle-list',
176         rasterizationState: {
177             frontFace: 'cw', // clock wise
178             cullMode: 'none', // don't cull faces
179         },
180         fragmentStage: {
181             module: device.createShaderModule({
182                 code: frag_shader_code_spirv
183             }),
184             entryPoint: 'main'
```

```
185         },
186         depthStencilState: {
187             format: 'depth32float',
188             depthWriteEnabled: true,
189             depthCompare: 'greater'
190         },
191         colorStates: [
192             {
193                 format: canvas_format,
194                 alphaBlend: {
195                     srcFactor: 'one',
196                     dstFactor: 'zero',
197                     operation: 'add'
198                 },
199                 colorBlend: {
200                     srcFactor: 'one',
201                     dstFactor: 'zero',
202                     operation: 'add'
203                 },
204                 writeMask: GPUColorWrite.ALL
205             }
206         ]
207     });
208 
209     var common_data = new Float32Array([
210         // view matrix
211         1.0, 0.0, 0.0, 0.0,
212         0.0, 1.0, 0.0, 0.0,
213         0.0, 0.0, 1.0, 0.0,
214         0.0, 0.0, 0.0, 1.0,
215         // proj matrix
216         1.0, 0.0, 0.0, 0.0,
217         0.0, 1.0, 0.0, 0.0,
218         0.0, 0.0, 1.0, 0.0,
219         0.0, 0.0, 0.0, 1.0,
220     ]);
221 
222     var common_buffer = initBuffer(common_data, GPUBufferUsage.UNIFORM, device,
        queue);
223 
224     var bind_group_common = device.createBindGroup({
225         layout: bgl_common,
226         entries: [
227             {
228                 binding: 0,
229                 resource: {
230                     buffer: common_buffer,
231                     offset: 0,
232                     size: common_data.byteLength
233                 }
234             }
235         ]
236     });
237 
238     var model_data = new Float32Array([
239         // model matrix (translate 2 units in Z direction)
240         1.0, 0.0, 0.0, 0.0,
241         0.0, 1.0, 0.0, 0.0,
242         0.0, 0.0, 1.0, 0.0,
243         0.0, 0.0, 0.0, 1.0
244     ]);
245 
```

```
246     var dynamic_buffer = initBuffer(model_data, GPUBufferUsage.UNIFORM, device,
            queue);
247
248     var bind_group_dynamic = device.createBindGroup({
249         layout: bgl_dynamic,
250         entries: [
251             {
252                 binding: 0,
253                 resource: {
254                     buffer: dynamic_buffer,
255                     offset: 0,
256                     size: model_data.byteLength
257                 }
258             }
259         ]
260     });
261
262     render(device, swap_chain, queue, canvas, render_pipeline, vertex_buffer,
            index_buffer, bind_group_common, bind_group_dynamic, index_data);
263
264 }
265
266 function render(device, swap_chain, queue, canvas, render_pipeline, vertex_buffer,
        index_buffer, bind_group_common, bind_group_dynamic, index_data) {
267     var color_texture = swap_chain.getCurrentTexture();
268     var depth_texture = device.createTexture({
269         size: {
270             width: canvas.width,
271             height: canvas.height,
272             depth: 1
273         },
274         mipLevelCount: 1,
275         sampleCount: 1,
276         dimension: '2d',
277         format: 'depth32float',
278         usage: GPUTextureUsage.OUTPUT_ATTACHMENT
279     });
280
281     var command_enc = device.createCommandEncoder();
282     var render_pass = command_enc.beginRenderPass({
283         colorAttachments: [
284             {
285                 attachment: color_texture.createView(),
286                 loadValue: {r: 0.3, g: 0.0, b: 0.2, a: 1.0}, // clear to dark purple
287                 storeOp: 'store'
288             }
289         ],
290         depthStencilAttachment: {
291             attachment: depth_texture.createView(),
292             depthLoadValue: 0.0,
293             depthStoreOp: 'store',
294             depthReadOnly: false,
295             stencilLoadValue: 0,
296             stencilStoreOp: 'store',
297             stencilReadOnly: true
298         },
299     });
300
301     render_pass.setPipeline(render_pipeline);
302     render_pass.setVertexBuffer(0, vertex_buffer);
303     render_pass.setIndexBuffer(index_buffer);
304     render_pass.setBindGroup(0, bind_group_common);
```

```
305     render_pass.setBindGroup(1, bind_group_dynamic);
306     render_pass.drawIndexed(index_data.length);
307     render_pass.endPass();
308
309     var command_buffer = command_enc.finish();
310     queue.submit([command_buffer]);
311
312     requestAnimationFrame(function() {
313         render(device, swap_chain, queue, canvas, render_pipeline, vertex_buffer,
                index_buffer, bind_group_common, bind_group_dynamic, index_data);
314     });
315 }
316
317 start();
```

Listing C.4: Kompletter JavaScript-Quellcode für das Minimalbeispiel aus Kapitel 4