# **W-Seminar 2024-2026 Algorithmen in Computerspielen**

# **Erster Bericht zum Arbeitsstand**

Matrixalgebra in der Computergrafik und ihre Anwendung in der OpenGL-Pipeline

Matthias Stößel

## **Beschreibung der inhaltlichen Schwerpunktsetzung**

Meine Arbeit zur Matrixalgebra in der Computergrafik und ihre Anwendung in der WebGL-Pipeline setzt sich zum Ziel mit Hilfe einer selbst programmierten kleinen Game Engine ein Spiel zu erstellen.

Dazu verwende ich die WebGL Libary und glMatrix. Damit erzeuge ich die benötigten Grafiken mithilfe von WebGL selbst.

Dafür benötige ich folgende Implementierungen:

* Grafikredering:
  + Mit der WebGL Pipeline sollen einfache Formen auf dem Bildschirm erzeugt werden können
* Mathematik für 2D-Transformationen:
  + Mit den Matrix Transformationen der Translation, Rotation und Skalierung sollen sich die Objekte bewegen
* Texturen:
  + Um ein eindrucksvolles Spielerlebnis zu gewährleisten brauchen die Objekte Texturen
* Kollisionsabfrage:
  + Damit man ein Spiel mit einem Punkte System programmieren kann braucht man einen Weg Punkte zu sammeln, oder das Spiel zu verlieren. Hierzu benötigt man Kollisionen.
* Input-Handling:
  + Damit das Spiel zu einem Spiel wird benötigt man einen Weg mit den Objekten zu interagieren. Dafür benötigt man eine Möglichkeit für Eingaben

Mit Hilfe dessen ist möchte ich ein kleines Spiel programmieren, in dem man mit einem Raumschiff Weltraumkörpern ausweicht. Es soll ein einfach zu implementierendes Spiel sein, da der Schwerpunkt eben auf der Game Engine liegt.

## **Geplante Gliederung der schriftlichen Arbeit**

1. **Einleitung:**
   1. Motivation und Zielsetzung
   2. Bedeutung der Matrixalgebra für die Spielentwicklung
   3. Vorstellung der 2D Game Engine
2. **Grundlagen der Matrixalgebra:**
   1. Definition und Operation mit Matrizen
   2. Transformation: Translation, Rotation, Skalierung
   3. Zusammensetzung der Transformationen
   4. Homogene Koordinaten und ihre Rolle in der Computergrafik
3. **Matrixalgebra in der Game-Engine Entwicklung:**
   1. Verwendung von Matrizen zur Objektplatzierung
   2. Implementierung von Bewegungen und Animationen
   3. Mathematische Optimierungen für Echtzeitanwendungen
4. **Architektur der 2D Game Engine:**
   1. Überblick über die Engine Struktur
   2. Redering-Modul: Verwendung von WebGL
   3. Kollisionsabfrage
   4. Benutzersteuerung: Integration von Eingabemethoden
5. **Begleitendes Programmierprojekt:**
   1. Ziel des Projekt: Entwicklung eines kleinen 2D-Spiels
   2. Beschreibung der Spielmechanik
   3. Umsetzung der Kernfunktionen der Engine (Redering, Transformation und Kollision)
6. **Herausforderungen und Lösungen:**
   1. Probleme bei der Implementierung
   2. Effiziente Nutzung von WebGL (Performance Optimierung)
   3. Debugging von Transformationen und Kollisionsmechaniken
7. **Fazit und Ausblick:**
   1. Zusammenfassung der Ergebnisse
   2. Mögliche Erweiterungen der Engine (z.B. Beleuchtung, Physik und Partikelsystem)

## **Gefundene Literatur**

#### **Interactive Computer Graphics: A Top-Down Approach with WebGL (Edward Angel, Dave Shreiner) 2014**

ISBN: 978-0133574845

Dieses Buch bietet eine umfassende Einführung in die Computergrafik mit Fokus auf WebGL. Es behandelt grundlegende Konzepte der Grafikprogrammierung, darunter Transformationen, Shader-Programmierung und die Pipeline-Architektur. Besonders hilfreich ist der detaillierte Abschnitt über die Anwendung von Matrixalgebra in 3D-Grafiken.

#### **WebGL Fundamentals (**[**https://webglfundamentals.org/**](https://webglfundamentals.org/)**)**

Diese Website bietet detaillierte Tutorials zur WebGL-Programmierung. Es wird Schritt für Schritt erklärt, wie Matrizen für Transformationen genutzt werden, und wie sie in Shadern implementiert werden. Der Fokus liegt dabei auf praxisnahen Beispielen.

#### „WebGL Turorial“ Playlist –Youtube (**https://www.youtube.com/watch?v=y2UsQB3WSvo&list=PLjcVFFANLS5w6Qbj\_1ziwT2LUHAwgZO2D**)

Diese Playlist bietet einen hervorsagend erklärten Einstieg in die Implementierung von WebGL. Sie beschreibt neben der Erstellung des klassischen Dreieck auch wie man in WebGl Koloriert und Bewegungen einbaut.

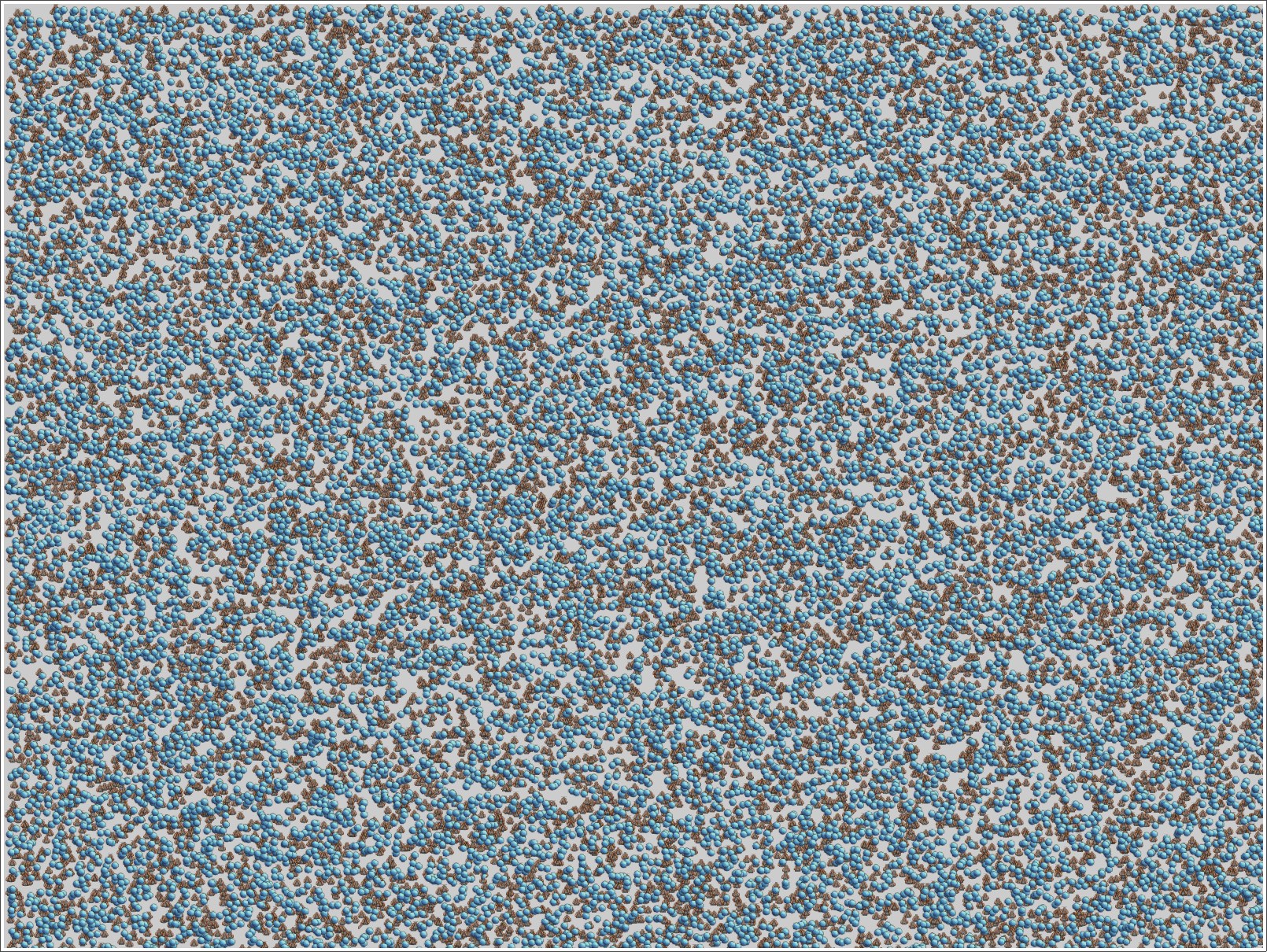
#### **MDN Web Docs – WebGL API (**[**https://developer.mozilla.org/en-US/docs/Web/API/WebGL\_API**](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API)**)**

Die offizielle Dokumentation der WebGL-API bietet eine solide Grundlage für die Programmierung mit WebGL. Sie enthält Beispiele für Transformationen und erklärt, wie Matrizen in die Shader-Pipeline integriert werden.

#### **Khronos Group – OpenGL und WebGL Spezifikationen (**[**https://www.khronos.org/**](https://www.khronos.org/)**)**

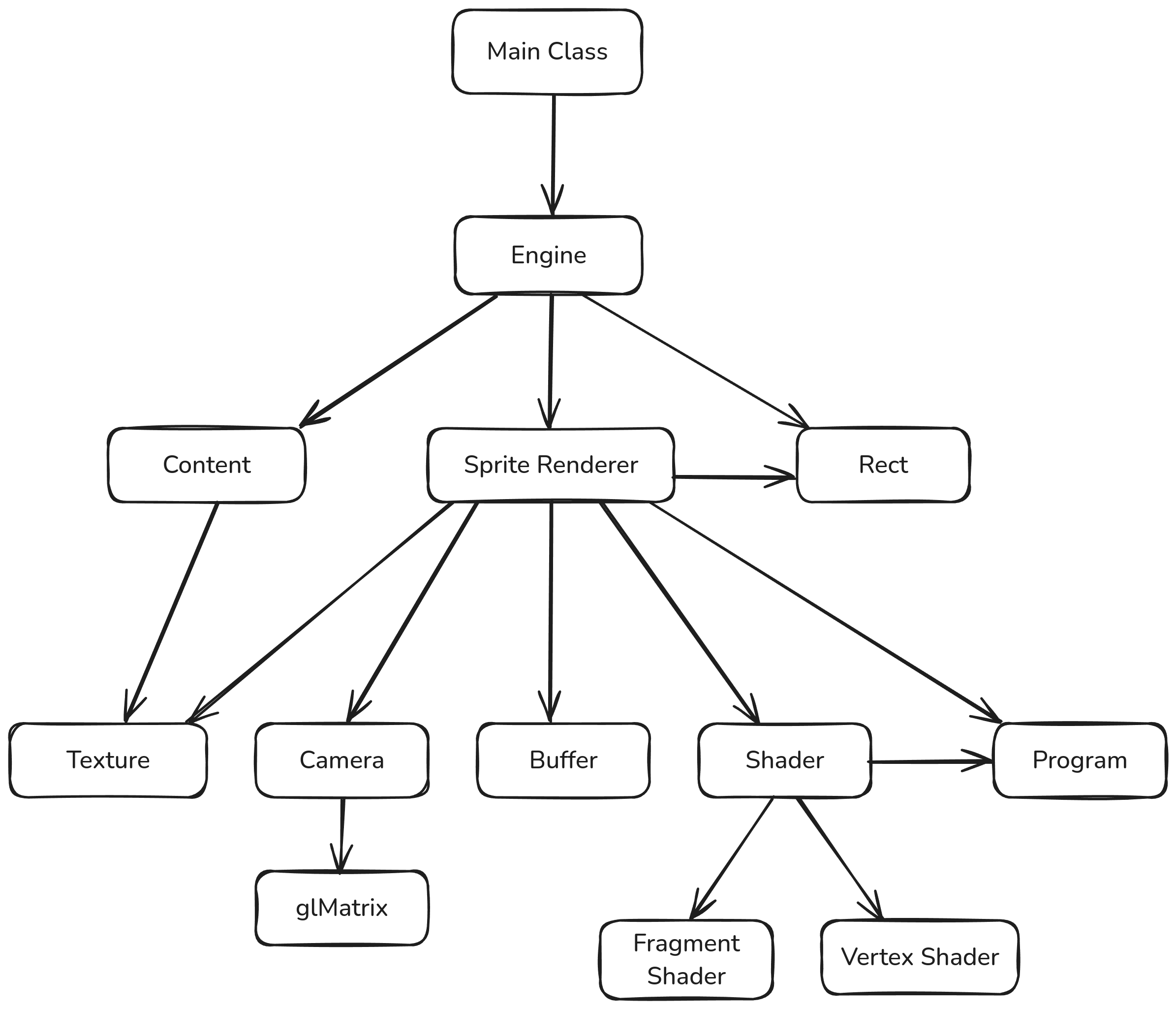
Die Khronos Group ist für die Spezifikationen von OpenGL und WebGL verantwortlich. Ihre Website bietet umfangreiche technische Details und Beispiele zur Implementierung der Grafikpipeline.

## **Begleitendes Programmierprojekt**



## Zu meiner theoretischen Erklärung werde ich wie bereits erwähnt eine Game Engine programmieren, mit deren Hilfe ich ein kleines 2D Weltraumspiel schaffen möchte.

Hierzu sehen sie in dem obigen Bild die erste Implementierung dieser Engine. Dabei werden mehrere tausend Sprites gerendert, um die Performance der Engine zu testen. Das dahinter stehende Programm ist wie folgt aufgebaut:



**Fragment Shader:**

Der Fragment Shader in WebGL berechnet die Farbe und andere Eigenschaften jedes Pixels (Fragments), das auf dem Bildschirm angezeigt wird. Er erhält Eingabedaten wie Farben, Texturen und Lichtinformationen und gibt die endgültige Farbe jedes Fragments zurück.

**Vertex Shader:**

Der Vertex Shader in WebGL verarbeitet die Geometriedaten eines 3D-Objekts, wie z.B. die Positionen der Eckpunkte (Vertices), und berechnet deren Transformation. Er erhält Eingabedaten wie Vertex-Positionen, Normalen und Texturkoordinaten und gibt die transformierten Positionen und andere Attribute für jedes Vertex zurück.

**Program:**

Das Program ist die Zusammensetzung vom Fragment und Vertex Shader die gemeinsam die Rendering-Pipeline durchlaufen. Es besteht aus zwei getrennten Shadern, die jeweils in GLSL (OpenGL Shading Language) geschrieben werden. Das Programm wird auf der GPU ausgeführt und bestimmt, wie die Geometrie eines Objekts (Vertex Shader) und die Farbe jedes Pixels (Fragment Shader) berechnet werden

**Buffer:**

Der Buffer in WebGL ist ein Speicherbereich auf der GPU, der Daten wie Vertex-Positionen, Texturkoordinaten oder Farbwerte speichert. Diese Daten werden vom CPU an die GPU übertragen, damit sie von den Shadern verarbeitet werden können.

**Camera:**

Eine Camera in WebGL definiert den Blickwinkel und die Perspektive, aus der die Szene gerendert wird. Sie legt fest, wie die 3D-Welt auf den 2D-Bildschirm projiziert wird. Die Kamera verwendet Parameter wie Position, Blickrichtung und Projektionsmatrix (z.B. Perspektive oder Orthogonalprojektion), um zu bestimmen, welche Teile der Szene sichtbar sind und wie sie dargestellt werden.

**Texture:**

Eine Textur in WebGL ist ein 2D-Bild oder eine Sammlung von Daten, die auf 3D-Objekte angewendet wird, um deren Oberfläche realistisch darzustellen. Texturen können Farben, Muster oder auch andere Informationen wie Normalen oder Transparenz enthalten. Sie werden im Fragment Shader verwendet, um die visuellen Details der Oberfläche eines Objekts zu berechnen. Texturen können auf Objekte projiziert werden, indem sie mit Texturkoordinaten (UV-Koordinaten) verbunden werden.

**Content:**

Diese Classe beinhaltet die verschiedenen Sprites die es Gibt. Dafür verwendet sie die Klasse Texture um die einzelnen Texture Assets zu laden.

**SpriteRenderer:**

Die Klasse Sprite Renderer sorgt dafür, dass mithilfe der untergeordneten Klassen letztendlich die Frames erzeugt werden.

**Rect:**

Die Klasse Rect dient dazu, dass das zeichnen von Sprites einfacher gemacht wird, indem man für den jeweiligen Sprite die Postion uns die Größe übergibt:

constructor(public x: number, public y: number, public width: number, public height: number)

## **Bisherige Schwierigkeiten**

Performance:

Nachdem ich die Grundlage für das Rendern der Sprites hatte, stellte ich fest, dass sobald ich mehrere Sprites Rendern wollte, die Frames pro Sekunde massivst darunter litten. Mit dem Browser Addon Spector.js konnte ich dann sehen, dass jeder Sprite einzeln gerendert wurde. Die Lösung lag nahe. Man muss alle Sprites auf einmal rendern können. Nach der implementierung des sogenannten Sprite Batchings wurden nun alle Sprites gleichzeitig gerenderet und die Frames pro Sekunde erholten sich wieder auf das gewünschte Niveau.