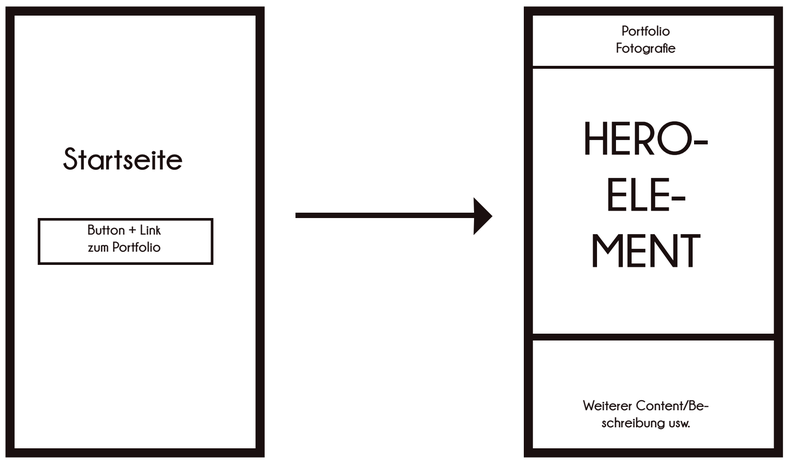
Konzept



Viele Designer und Front-end Entwickler besitzen eigene Portfolio Seiten. Diese Seiten sind besonders dafür geeignet, bei möglichen Arbeitgebern oder Kunden Interesse zu wecken. Ein Schlüsselpunkt dieses Interesses ist es, dem Nutzer eine Erfahrung zu bieten, die sich von seinen sonstigen Interaktionen im Netz abhebt. Allgemein ist ein großes Problem im Internet die Aufmerksamkeitsspanne, mit der sich Nutzer durchs Netz bewegen, im Online-Marketing Bereich spricht man von der sog. „Verweildauer". Portfolio-Websiten sind natürlich ein recht spezifischer Anwendungsfall und Dinge, die hier gelten, können wohl nicht allgemein auf UI/UX Design angewendet werden, jedoch in dem spezifischen Fall meiner eigenen Seite wollte ich wie gesagt eine Möglichkeit schaffen, eine individuelle Erfahrung entwickeln zu können. Wichtige Aspekte hierfür sind die funktionale Darstellung meiner Fähigkeit im Umgang mit Kameras und außerdem eine spannende Interaktionsmöglichkeit. Three Js bietet eine Möglichkeit, Design sowohl funktional als auch interessant umzusetzen.

THREEJS

Bevor man Shader-Code ausführen kann, muss man noch einige Dinge bezüglich THREEJS beachten. Allgemein basiert THREEJS auf einem JS Package namens WebGl. THREEJS versucht die komplexen mathematischen Funktionen, die 3D Programmierung zugrunde liegen, etwas zur vereinfachen. Bevor man sich dem Shader widmen kann muss noch eine allgemeine Szene in THREEJS aufgesetzt werden.
Eine Boilerplate dafür findet man hier: https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene
Es gibt mehrere Arten von Shadern. Da der Shader-Code aber selbst geschrieben werden sollte, wird hier auf die „rohste" Form zurückgegriffen. Der Shader besteht aus 2 Komponenten, einem Vertex- und einem Fragment Shader. Der Vertex Shader beeinflusst die eigentliche Geometrie des Objekts und der Fragment Shader mehr oder weniger deren Oberfläche. THREEJS basiert auf Javascript unsere Shader basieren auf einer Sprache namens GLSL.

```
function fragmentShader(){
return `
Shadercode
`}
```

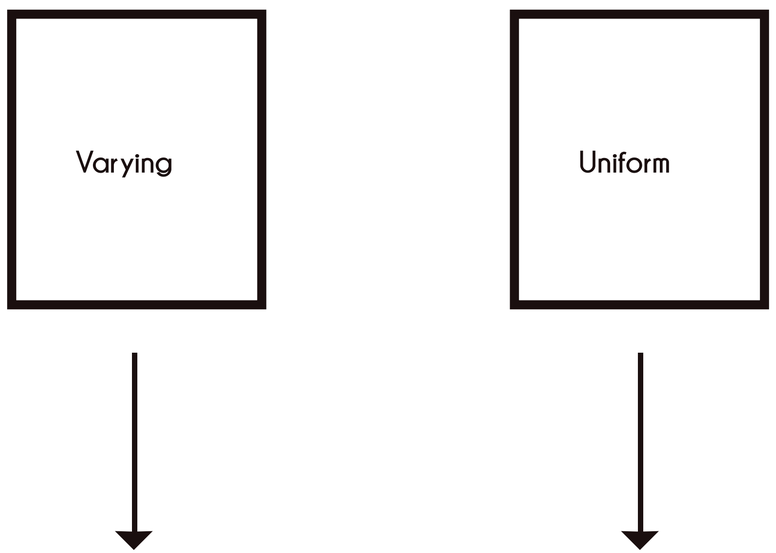
Damit die Grafikeinheit unseres Rechners den Shader direkt ausführen kann und er trotzdem Teil eines Javascript-Projekts bleibt, besteht die Möglichkeit den Shader direkt als String einer Funktion zurückgeben zu können.

```
let cubematerialaterial =  new THREE.ShaderMaterial({
uniforms: seconduniforms,
fragmentShader: cubefragment(),
vertexShader: vertexShader(),
wireframe:false,
});
var cubemesh = new THREE.Mesh(cube,cubematerialaterial);
scene.add(cubemesh);
`}
```

Diese Shader werden dann einem Material hinzugefügt, das Material wird zusammen mit einer Geometrie dann in ein Mesh umgewandelt und dann erst kann es der Szene hinzugefügt werden, um es zu rendern.

GLSL

Der eigentliche Code für die Shader fängt eigentlich erst ab Zeile 44 an. Zunächst kommt der Vertex Shader. Der Vertexshader ist in diesem Fall nicht sonderlich komplex. Es gibt sowohl für den Vertex Shader, als auch für den Fragment Shader Code, der definitiv vorhanden sein muss, damit das Ergebnis funktioniert. Warum genau liest man vielleicht lieber in den dementsprechenden Dokumentationen nach (https://de.wikipedia.org/wiki/OpenGL_ES_Shading_Language). Was höchstens auffällig sein könnte, ist dass wir die 2D Repräsentation unseres 3D Objekts (die sog. UV Map) abhängig machen zur Position der Maus. Diese UV Map wird dann auch an den Fragment Shader weiter gegeben.



Neben den üblichen Datentypen wie BOOLEAN, STRING, FLOAT etc. gibt es spezifische Datentypen die in der Shaderprogrammierung angewendet werden.

Für dieses Projekt waren Uniforms besonders interessant, da sie die Möglichkeit geben Events ab zu fangen und die Shader darauf reagieren zu lassen.
Zu Beginn müssen Uniforms als Array initialisiert werden.

```
let uniforms = {
u_X:{type:'float', value: 1.0},
u_Y:{type:'float', value: 1.0},
u_resolution: { type: „v2", value: new THREE.Vector2()},
u_Time:{type:'float', value: 0.2},
u_smoothedtome: {type:'float', value:0.4},
}
```

Insgesamt gibt es 2 Fragment Shader in dem gesamten Projekt. Der erste Shader projeziert Bilder auf einen Cube. Und verzerrt diese dann zu einem Mouse Event.

```
float noise (in vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);

// Four corners in 2D of a tile
    float a = random(i);
float b = random(i + vec2(1.0, 0.0));
float c = random(i + vec2(0.0, 1.0));
float d = random(i + vec2(1.0, 1.0));

// Smooth Interpolation

vec2 u = f*f*(3.0-2.0*f);
```

Diese Noise Funktion habe ich direkt aus dem Book of Shaders übernommen. Im Endeffekt ist sie nur dafür da , die Werte, die rein gegeben werden, abstrahiert wieder zurückzugeben. Floor wandelt die Werte von einem Float in einen Integer um. die random() function benutzt 2 Funktionen, um die ausgegebenen Werte wieder zu randomisieren. Die „Frac“ function ist eine in GLSL verfügbare Funktion. Diese kann benutzt werden, um die Zahlen hinterm Komma eines Floats zu erhalten. „Sin“ ist einfach eine Sinus Funktion und „dot“ liefert das Skalarprodukt. Auf diese Funktion wäre ich selbst nie gekommen, aber da sie im Book of Shaders, als Beispiel für eine randomisierungsfunktion gegeben wurde, habe ich sie einfach übernommen.

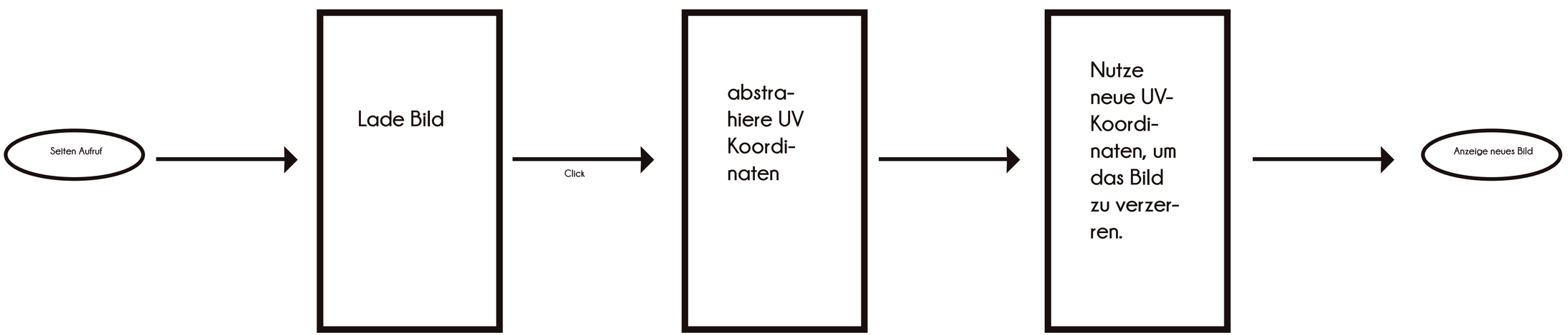
```
float random (in vec2 st) {
    return fract(sin(dot(st.xy,
            vec2(12.9898,78.233)))
            * 43758.5453123);
        }
```

```
if(trigger==true){
float Zeitsmooth= u_Time*0.05;
meshdimension.x=noise(meshdimension)/Zeitsmooth;
meshdimension.y=noise(meshdimension)/Zeitsmooth;
vec4 miximage= texture2D(texture2,vUv);

meshdimension.y= mix(meshdimension.y,miximage.r/Zeitsmooth, 0.7);
meshdimension.x= mix(meshdimension.x,miximage.r/Zeitsmooth, 0.7);

secondimage=  vec4(mix(texture2D(texture2, meshdimension),miximage,Zeitsmooth));
}
```

Sobald das Click-Event aktiviert wird, werden die UV Koordinaten des Meshes verzerrt. Da der Zeitfaktor etwas zu groß war, wird er mit 0.005 multipliziert.



Auch der zweite Fragment Shader beginnt mit einer Funktion aus dem Book of Shaders. Diese erzeugt einen Kreis.

```
float circleShape(vec2 position, float radius, vec2 Mouse)
{
return step(radius, length(position-Mouse));
}
```

Danach folgt eine Funktion, die ich ebenfalls aus dem Book of Shaders übernommen habe, auch hier dient sie einfach nur zur Randomisierung.

```
//Make more Circles
// ---------------------------------------------------------------------------------------------------------------
pattern=step(sin(st.x+bewegungsfaktor*0.005)+tan(st.y/bewegungsfaktor)+cos(st.y/bewegungsfaktor),0.2);
vec2 makemousealittlesmoother= vMouse*0.5;
float secondcircle = circleShape(vec2(st.x,st.y),0.05,vec2 (0.4+makemousealittlesmoother.x, 0.07+makemousealittlesmoother.y));
vec3 secondcircleshape= vec3( secondcircle+ pattern);

combined=secondcircleshape*combined;
```

Anschließend werden Mehrere Kreise erzeugt, die sich zur Maus bewegen. Außerdem wird ein Muster auf die Kreise rauf addiert.

```
//Putting it all together
// ---------------------------------------------------------------------------------------------------------------
combined= vec3 (1.-combined.r,1.- combined.g, 1.-combined.b);
gl_FragColor = vec4(combined, 1.0 );
//Putting it all together
// ---------------------------------------------------------------------------------------------------------------
```

Hier wird dann die finale Farbe fest gelegt.

Fazit

Der Hauptpunkt der Aufgabe ist mir wohl geglückt, der Glitch Effekt zwischen den einzelnen Bildern und ein geneartives Muster, das abhängig ist zur Zeit und der Maus. Allerdings muss ich sagen, dass mir ein paar Dinge dann doch erstaunlich schwerfielen. Generell wurde es durch den selbst gewählten Anwendungsbereich nicht unbedingt leichter. Ich hätte mich wohl für das Projekt lieber nur auf OPENGL/ GLSL konzentrieren sollen. Dann wäre aber die Möglichkeit weggefallen, das Ganze für meine Website nutzen zu können. Alleine der Zugriff auf die Laufzeit war erstaunlich schwer, da das übliche Javscript Prozedere mit getDate(); ins unendliche hoch zählt und das ganze über Modulo zu lösen, nicht hinreichend funktioniert hat. Auf die Idee, eine externe Libary für die Timeline zu nehmen, bin ich erst recht spät gekommen. Des Weiteren wusste ich zwar, dass ich einen Effekt für ein Bild haben wollte, aber das ganz ursprüngliche Vorbild https://tympanus.net/codrops/2019/02/20/how-to-create-a-fake-3d-image-effect-with-webgl/ wurde in vanilla WEBGL geschrieben und die Übersetzung auf THREEJS war am Ende doch schwerer als gedacht.
https://codepen.io/ReGGae/pen/bmyYEj war deswegen dann die neue Vorlage, allerdings habe ich versucht, das Ganze mit dem vorherigen Beispiel zu vermischen und dem Ganzen noch etwas Eigenes bei zu steuern.
Generell ist Shaderprogrammierung eine sehr neue Art der Programmierung gewesen, weswegen ich wohl mit einigen Aspekten erst einmal komplett überfordert war. Ich bin zwar recht fit in MAYA und BLENDER, aber diese Art der Programmierung war für mich komplett neu. Außerdem bin ich bei Weitem nicht so fit in Mathe, wie es für diese Art der Programmierung wohl von Vorteil gewesen wäre.
Trotz allem bin ich froh, mich damit auseinandergesetzt zu haben. Ich finde es prinzipiell spannend, neben den klassischen Front-End Möglichkeiten noch Shaderprogrammierung zur Gestaltung nutzen zu können, aber dafür bedarf es wohl definitiv mehr Übung.