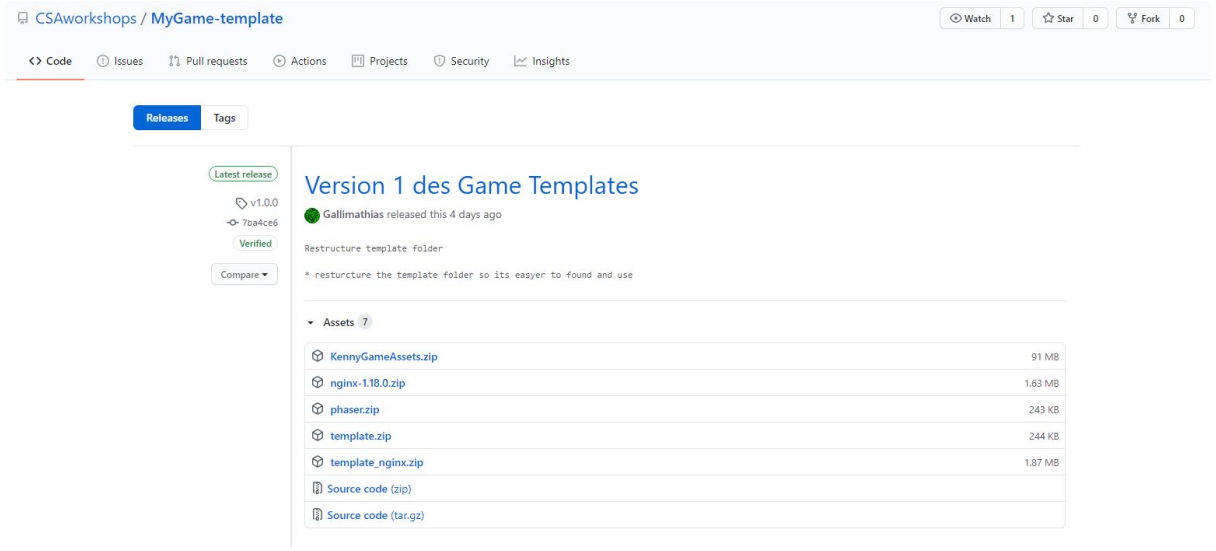
Programmier-Workshop: Gamedesign mit Javascript

mit Maximilian Krüger

1. Setup und Grundlagen

Alle wichtigen Downloads gibt es auf
https://github.com/CSAworkshops/MyGame-template/releases



a. Serversetup

1. *nginx-1.18.0.zip* aus dem Github herunterladen
2. Lege dir an einem beliebigen Ort (z.B. Desktop oder “Dokumente”) einen Ordner an, z.B. “Workshop” und verschiebe die Zip Datei aus deinem Download Ordner in den neuen Ordner (“Workshop”)
3. Per Rechtsklick auf die Datei die Datei in den eigenen Ordner entpacken (extrahieren)
4. *nginx.exe* per Doppelklick starten und angeforderte Zugriffe erlauben
5. Gearbeitet wird nun in dem Ordner: *nginx-1.18.0 > html*

b. Servertest

Durch Eingabe von “localhost” in einem Browser deiner Wahl (Google Chrome, Firefox, Edge) gelangt man auf die Startseite von nginx.
In der Datei *nginx-1.18.0 > html > index.html* findet sich der HTML Code der nginx Startseite, in der rumprobiert und die Startseite nach eigenem Wunsch verändert werden kann.

Hilfe für Programmieren mit HTML: https://www.w3schools.com/

c. Spiel Setup

1. *Phaser.zip* aus dem Github herunterladen
2. Die Datei im selben Ordner wie zuvor (“Workshop”) entpacken.
3. den untersten “Phaser” Ordner per Rechtsklick oder STRG+X ausschneiden und in den Ordner *nginx-1.18.0 > html* (wo auch die *index.html* Datei liegt) einfügen
4. In der *index.html* Die Inhalte von “<style>” und “<body>” entfernen und in <body> folgenden Zeilen einfügen:

   <script src=”phaser/phaser.min.js”>
   <script src=”game.js”></script>.

   Der Titel in <title> kann zudem durch einen selbstgewählten Spieletitel ersetzt werden.

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4   <title>Action mit der Feuerwehr</title>
5   <meta charset="utf-8"/>
6   <script src="phaser/phaser.min.js"></script>
7   </head>
8   <body>
9       <script src="game.js"></script>
10  </body>
11  </html>
12
```

5. Im Ordner *nginx-1.18.0 > html* eine neue Datei mit dem Namen “game.js” anlegen.
   Durch Eingabe eines Befehls (z.B. *alert(‘Tatütata’);*), einem Abspeichern der *game.js* und Neuladen der “localhost” Webseite kann die Funktionsfähigkeit getestet werden.

d. Arbeitsumgebung

Für die weitere Spieleprogrammierung werden folgende Tools empfohlen, alternativen nach eigenen Vorlieben sind möglich.

https://notepad-plus-plus.org/downloads/
https://code.visualstudio.com/

e. Javascript Grundlagen

Variablen-Deklaration:

```
let sound = "Tatütata";
```
Hier wird der Variable *sound* der Wert “Tatütata” zugeordnet
Statt *let* kann auch *var* verwendet werden, es gibt jedoch Unterschiede im Gültigkeitsbereich.

Ausgabe:

```
alert(sound);
```
Der Befehl *alert* gibt den Inhalt der Klammern (hier *sound*) aus

Funktionen:
Eine eigene Funktion kann mit dem Befehl *function* und einem selbstgewählten Namen im Anschluss erstellt werden. Der Inhalt dieser Funktion kann dann mit dem Namen der Funktion plus Klammer auf und Klammer zu aufgerufen werden.

```
CallTheFirefighters();


function CallTheFirefighters(){
    alert(sound3);
}
```

Wird in der Klammer eine Variable angegeben, kann der Funktion auch ein Parameter übergeben und in der Funktion verarbeitet werden. Genauso kann durch ein *return* am Ende der Funktion ein Wert an den Aufruf der Funktion zurückgegeben werden

```
let final = CallTheFirefighters(sound);
alert(final);

function CallTheFirefighters(soundName){
    let result = soundName + " 🚒";
    return result;
}
```

Arrays:
Arrays speichern in einer Variable mehrere Werte. Mit den Indizes von 0 startend können diese Werte geändert oder ausgelesen werden.

```
calls = ["Feuer", "kein Feuer", "Katze", "Blau"];
```

0 1 2 3

Mit dem Befehl *[arrayname].includes* kann die Bedingung abgefragt werden, ob sich ein Element in dem Array befindet.

```
let incomingCall = calls[i];

if(allowed.includes(incomingCall)){
```

Schleifen:
For-Schleifen agieren als Zähler und können beispielsweise Arrays bequem auslesen.

```
for(let i = 0; i < 4; i++){
    let final = CallTheFirefighters(calls[i]);
    alert(final);
}
```

While-Schleifen laufen, solange eine angegebene Bedingung erfüllt ist. Hier im Beispiel läuft die Schleife endlos.

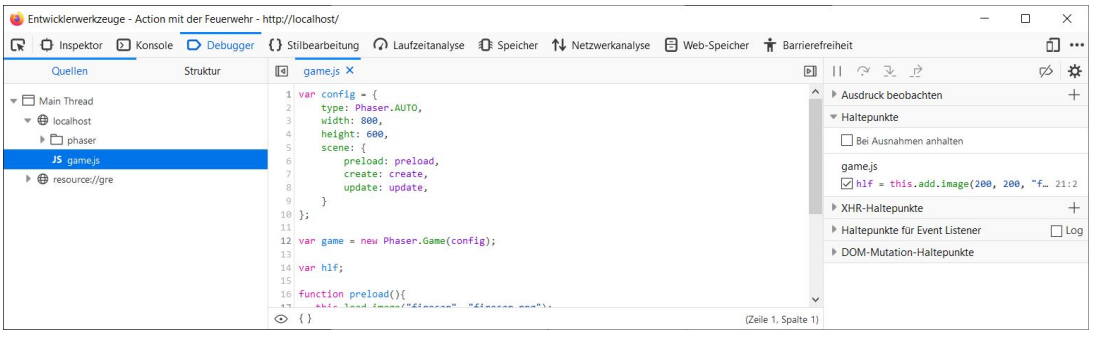
```
while(calls[0] == "Feuer"){
    alert("Test");
}
```

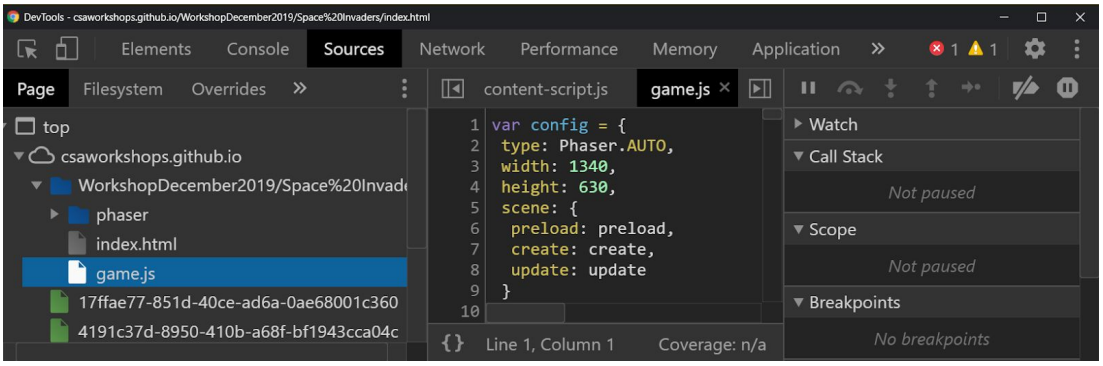
Bedingungen:
der If-Befehl dient dazu, folgende Anweisungen nur unter einer bedingung auszuführen. *else* und *else if* arbeiten weitere Fälle ab.

```
if(incomingCall == "Feuer"){
    let final = CallTheFirefighters(incomingCal
    alert(final);
} else if(incomingCall == "Katze") {
    alert("
");
}
```

f. Debugging

Mit der Taste F12 oder STRG + UMSCH +i werden im Browser die Entwickleroptionen angezeigt. In dem Tab Konsole werden dann Fehler angezeigt und zeigen oft eine Lösung bei bestehenden Problemen an.





2. Tag 1

Links zu den Ergebnissen vorheriger Workshops

Dezember 2019: https://csaworkshops.github.io/WorkshopDecember2019/
Mai 2020: https://csaworkshops.github.io/WorkshopMai2020/

a. Start der game.js

Zu Beginn werden in der *game.js* die wichtigsten Konfigurationsdaten mit Hilfe des Phaser Frameworks definiert.

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    scene: {
        preload: preload,
        create: create,
        update: update,
    }
}
```

Dazu werden die benötigten Funktionen sowie neuen Game Objektes erstellt um das Grundsetup abzuschließen. Damit wird im Browser zumindest mal

eine schwarze Box angezeigt.

```
    scene: {
        preload: preload,
        create: create,
        update: update,
    }
};

var game = new Phaser.Game(config);

function preload(){

}

function create(){

}

function update(){

}
```

b. Recherche

Vor wildem drauf los Coden und vielem vor und zurück basteln ist gute Recherche das A und O. Die Überlegung zu Beginn sollte sich auf ein Thema fokussieren, welches man mit seinem Spiel umsetzen möchte. Danach ist Google dann dein bester Freund und Helfer: Welche Inspirationen kann man finden? was für Features soll mein Game haben? Welche Fähigkeiten habe ich um meine Ideen umzusetzen?

Anregungen zu Spielen basierend auf Phaser findet man hier: https://phaser.io/

Tipp: https://www.DeepL.com kann neben Google Translate sehr gut bei Übersetzungsschwierigkeiten helfen.

c. Grafiken

Damit das Endergebnis auch schön aussieht, gehören natürlich immer Grafik Assets dazu. Doch Achtung: Wer es nicht selber gestalten will und Google nutzt muss auf Lizenzbestimmungen beachten! Als Möglichkeit kann man in der Google Suche nach Lizenzfreien Bildern filtern.

Zusätzlich gibt es im Git Repository (Link ganz oben) eine Datei namens *KennyGames.zip* mit vielen Assets die benutzt werden können.

Alle benötigten Assets werden ebenfalls in den Ordner *nginx-1.18.0 > html* verschoben.

Weitere lizenzfreie Grafiken:
https://www.feathericons.com

d. Planung

Um die eigene Idee besser umsetzen zu können, hilft eine Visualisierung. Dazu kann man Papier und Stift nutzen oder digitale Programme wie https://www.draw.io. Eine einfache Konzeption ist leichter umzusetzen, als alles aus dem Kopf abzuarbeiten. Auch ein paar Stichworte zu Funktionen und Spielprinzipien können hilfreich sein.

e. Umsetzung

Um die ausgesuchten Grafiken auch im Spiel verwenden zu können, müssen die Bilder vor dem Spielstart geladen werden. Dazu wird in der *preload* Funktion folgender Befehl genutzt:

```
function preload(){
    this.load.image("firecar", "firecar.png");
}
```

Die Darstellung im Browser folgt im Anschluss in der *create* Funktion.

```
function create(){
  this.add.image(200, 200, "firecar");
}
```

Damit im Verlauf des Spieles das Bild verändert, sprich gedreht, bewegt und verändert werden kann, muss das Bild in einer Variablen abgespeichert werden. Dazu wird der letzte Befehl leicht abgeändert:

```
function create(){
  hlf = this.add.image(200, 200, "firecar"
}
```

Infos zu den Funktionen *preload*, *create* und *update*:
preload und *create* werden zu Beginn des Spieles einmalig aufgerufen, *update* hingegen wird per Schleife durchgehend aufgerufen (einmal pro Frame)

Um ein Objekt (hier Bild) zu bewegen, können in der update Funktion die Position des Objektes kontinuierlich angepasst werden.

```
function update(){
    hlf.x += 1;
}
```

Da eine einfache Bewegung ja langweilig ist, soll nun die Bewegung per Eingabe gesteuert werden. Dazu erstellen wir zuerst ein weiteres Objekt zur Steuerung in der *create* Funktion.

```
function create(){
  hlf = this.add.image(-110, 200, "firecar");
  cursors = this.input.keyboard.createCursorKeys();
}
```

Danach wird in der update Funktion geprüft, ob man die gewollte taste gedrückt hat und dementsprechend das Bild bewegt.

```
if(cursors.right.isDown){
    hlf.x += 1;
}
```

Infos zum Debugging:
Durch den Inspektor der Website (Taste F12) können Befehle Schritt für Schritt ausgeführt sowie Informationen zu Funktionen und Variablen angezeigt werden. Besonders die Markierung von Fehlern sowie Hinweistexte zu diesen können bei der Behebung äußerst hilfreich sein.

Nun kann man das Bild in der Browseranwendung durch Drücken der rechten Pfeiltaste nach rechts bewegen. Diese Bewegung wirkt jedoch sehr unecht, weshalb wir ein Physik Modell zum Spiel hinzufügen:

```
physics: {
  default: "arcade",
  arcade: {
      gravity: {
        y: 200,
      },      You, a few seconds ago • Uncommitted changes
  }
},
```

```
function create() {
  hlf = this.physics.add.image(200, 200, "firecar");
  cursors = this.input.keyboard.createCursorKeys();
  speed = 0;
  step = 20;
}

function update() {
```

Die Eingabe wird dementsprechend angepasst, um das Bild zu beschleunigen und zu bremsen.

```
function update(){
    if(cursors.right.isDown){
        hlf.setVelocityX(10);
    }else if(cursors.left.isDown){
        hlf.setVelocityX(-10);
    }
}
```

Globale Variablen werden daraufhin eingefügt, um die Beschleunigung/Abbremsung abzuschließen.

```
function create(){
  hlf = this.physics.add.image(200, 200, "firecar");
  cursors = this.input.keyboard.createCursorKeys();
  speed = 0;
  step = 20;
}
```

```
function update(){
    if(cursors.right.isDown){
        speed += step;
    }else if(cursors.left.isDown){
        speed -= step;
    }

    hlf.setVelocityX(speed);
}
```

Damit das Bild nicht zu schnell bewegt wird, werden Bedingungen eingefügt, die die Geschwindigkeit begrenzen.

```
if(speed > 150)
{
    speed = 150;
} else if(speed < -30){
    speed = -30;
}
```

Auch eine sofortige Stoppfunktion wird integriert, die per Leertaste ausgelöst wird.

```
function update(){
    if(cursors.right.isDown){
        speed += step;
    }else if(cursors.left.isDown){
        speed -= step;
    }

    if(cursors.space.isDown)
    {
        speed = 0;
    }

    if(speed > 150)
    {
        speed = 150;
    } else if(speed < -30){
        speed = -30;
    }
                        var speed: any
    hlf.setVelocityX(speed);
}
```

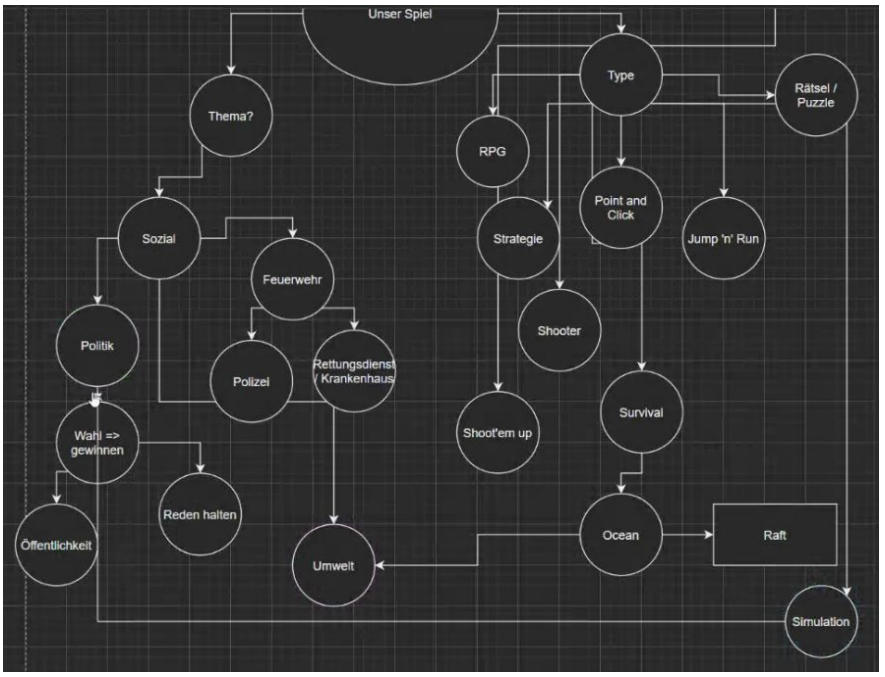
3. Tag 2

a. Findung eigener Spielidee

Wie bereits am Vortag werden im Brainstorming alle möglichen Ideen gesammelt. Hierbei gilt:

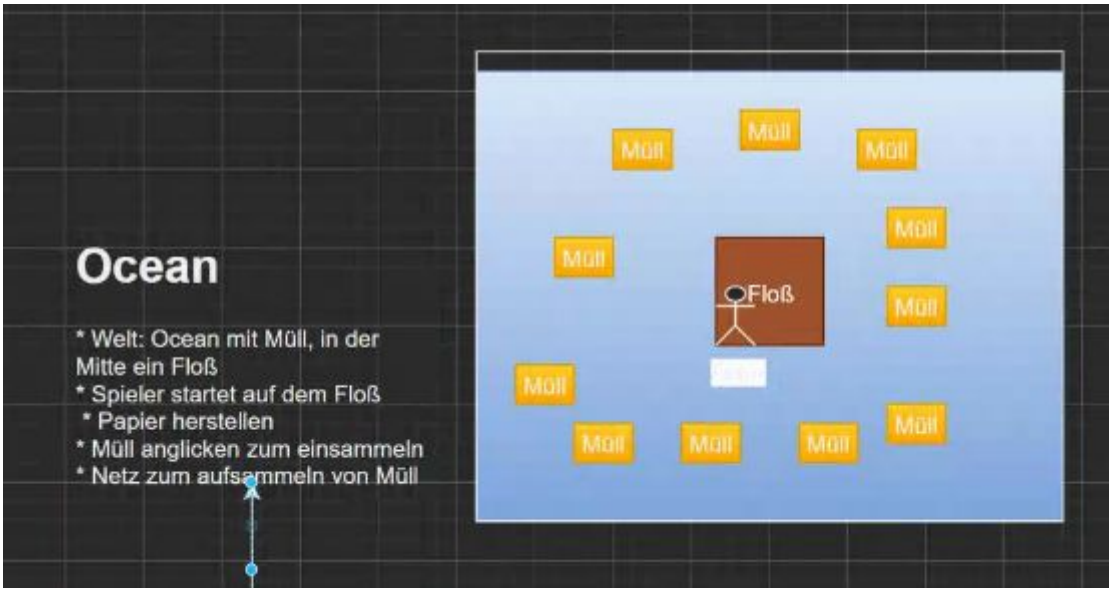
- je visueller desto besser
- keine Idee ist blöd
- keine Idee ist nicht umsetzbar

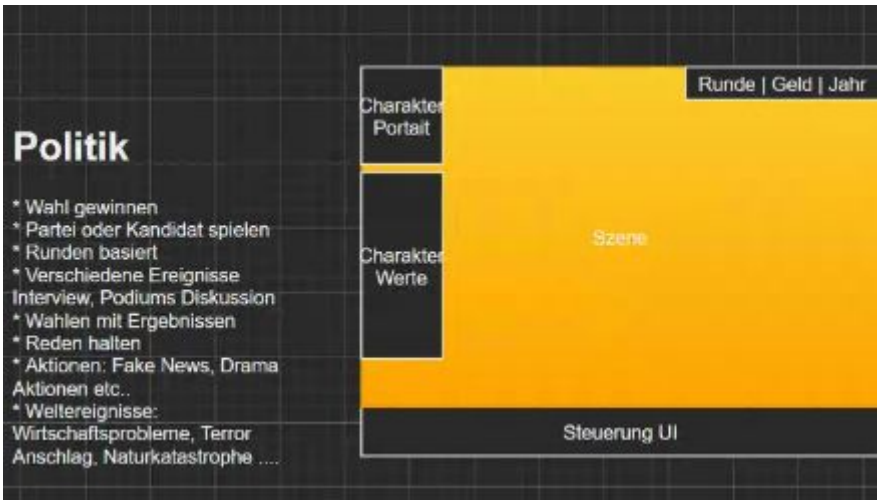
Hilfreich kann zudem eine Spiele Artenübersicht sein, um die grundlegenden Spielmechaniken einzugrenzen. Auch ein gewünschtes Thema sollte man sich überlegen, welches sich durch das gesamt Spiel zieht.



b. Ideen Festlegung

Um das Spielkonzept einzugrenzen werden nach und nach Ideen verfeinert, sich auf ein Thema und Spieltyp festgelegt. Weiters Brainstorming für genauere Spielmechanismen helfen dabei, sich ein klares Bild von dem Spiel zu machen.





Möglich ist auch, eine kleine Zahl von Grundideen nebeneinander auszuführen um Vor und Nachteile auszuarbeiten.

Steht dann eine grobe Auswahl zum Spielkonzept kann eine Liste von TODOS erstellt werden, die im Laufe der Spieleentwicklung abgearbeitet werden.

4. Tag 3

a. Hinweis

Damit aktuelle Version der *game.js* benutzt wird, sollte in den Entwickleroptionen (erreichbar per Taste F12) cache disabled werden

b. Sprites

Damit Objekte nicht nur als einzelnes Bild sondern als Animation angezeigt werden, wird eine Art Daumenkino Effekt genutzt. Dazu wird statt einem einzelnen Bild eine Bildserie geladen.

```
function preload() {
  this.load.spritesheet("firecar", "firecar_sprite.png")
}
```

Verwendet als Grafik wird weiterhin eine Bilddatei, jedoch enthält sie alle Stufen der Animation aneinander gehängt. Diese Stufen werden dann per Code in die einzelnen Teilbilder getrennt.

```
function create() {
  hlf = this.physics.add.sprite(200, 200, "firecar");
```

Weitere wird in der *create()* Methode folgendes benötigt um die Animation zu definieren und abzuspielen:

```
animation = this.anims.create({
  key: "blue",
  frames: this.anims.generateFrameNumbers("firecar", { start: 0, end: 3 }),
  frameRate: 6,
  repeat: -1
});

this.anims.play("blue", hlf);
```

Um Animationen spannender zu gestalten, und den Spieler selbst zu involvieren, können Animationen auch während des Spiels gesteuert (an- und ausschalten) werden.

Disclaimer: Um Keyboard-Inputs ab zu fragen, gibt es viele Methoden, hier wird beispielshalber eine Möglichkeit vorgestellt.

Zur Initialisierung der “Abhörmethode” wird in der *create()* Funktion folgende Zeile eingefügt:

```
this.input.keyboard.on("keydown-E", startBluelight.bind(this));
```

Die hier aufgerufene Funktion muss natürlich im Anschluss definiert werden und enthält die Aktionen, die bei Tastendruck ausgeführt werden sollen.

```
function startBluelight(event){
  this.anims.play("blue", hlf);
}
```

Damit aber nicht genug, denn eine Animation soll ja nicht nur einmal gestartet werden sondern immer wieder an und ausgeschaltet werden. dazu benötigen wir eine globale Variable, die den Status der Animation abspeichert:

var bluelight = false

dies kann dann in der eigenen Funktion verarbeitet werden und je nach aktuellem Status der Animation in den anderen Status zu wechseln.

```
function startBluelight(event) {
  if (bluelight) {
    animation.pause();
    bluelight = false;
  } else {
    this.anims.play("blue", hlf);
    bluelight = true;
  }
```

c. Eventsystem

Ein Punktesystem braucht genügend Überlegung um festzulegen wofür wie viele Punkte vergeben werden. Eine gute Balance hier kann ein Spiel deutlich spannender und unterhaltsamer machen. Auftauchende Events können da ein gutes Mittel sein.

Wir starten mit der Anzeige eines einfachen Textfeldes:

global:
var missionText

In create():

```
missionText = this.add.text(20, 20, "Mission: -", {
  fontSize: "20px",
  fill: "#FFFFFF",
});
```

Dazu werden Events hinzugefügt, die später zur Bepunktung dienen. In zeitlichen Abständen mit einer Zufallszeit wird dann der Text abgeändert.

```
this.time.addEvent({
  delay: Phaser.Math.Between(5000, 10000),
  callback: createMission.bind(this)
});
```

Dazu die aufgerufene Methode:

```
function createMission(event){
  missionText.text = "Mission: Es brennt";
}
```

d. Punktesystem

Die Punkteanzeige wird nun in ein Textfeld implementiert. Dazu benötigt man zu Beginn eine dedizierte Variable zum Speichern des Punktestandes.

var points = 0;

Dazu wird ein neues Textfeld erzeugt.

```
pointText = this.add.text(0, config.height - 20, points, {
  fontSize: "15px",
  fill: "#FFFFFF",
});
```

Um die Punkte dynamisch ändern zu lassen benötigt man in der update() Funktion folgenden Befehl:

```
pointText.text = points.toString();
hlf.setVelocityX(speed);
```

die points Variable kann dann an beliebiger Stelle um einen gewünschten Wert erhöht werden.

points += 100;

5. Tag 4

a. Klassen

Bisher wurde jeglicher Code nur in einer Datei geschrieben, dadurch endet mit man mit einem riesigen, unübersichtlichen Haufen an Code. Um das

Ganze aufzuarbeiten und auch in unterschiedliche Klassen zu unterteilen. Dazu kann man in demselben Ordner, in dem sich bereits unsere *game.js* befindet, neue .js Dateien anlegen.

Gerade Objekte im Spiel können so sehr strukturiert angelegt werden. Hier legen wir das bisher verwendete Objekt nun als Klasse an. Eine Klasse hat den Vorteil das man viele Objekte von dieser Klasse erzeugen kann.

```
class FireTruck{
    gameObject = {};
    animation= {};
    speedSteps= 120;
    start= () => {
      hlf.gameObject.setVelocityX(hlf.speedSteps);
    };
    stop= () => {
      hlf.gameObject.setVelocityX(0);
    };
    update= () => {

    };
}
```

Zum Besseren Verständnis:
Z.b. sind du und ich so gesehen ein Objekt und haben beide die Klasse Mensch. Alle Menschen haben die Gleiche Eigenschaften und Funktionsweisen aber trotzdem sind alle Menschen individuell und haben eigene zustände.

Damit wir unsere Klasse verwenden können. MÜssen wir ein Objekt von der Klasse erstellen

den Befehl zur Objekterstellung in der *create()* Funktion der *game.js*:

```
hlf = new FireTruck();
```

Zudem müssen wir die neue Datei in der *index.html* einbinden.

```
<!DOCTYPE html>
<html>
<head>
<title>Action mit der Feuerwehr</title>
<meta charset="utf-8"/>
<script src="phaser/phaser.min.js"></script>
</head>
<body>
    <script src="scripts/firetruck.js">
    <script src="game.js"></script>
</body>
</html>
```

Wir können unserer Klasse zudem noch einen sogenannten Konstruktor hinzufügen. Z.b. der FireTruck klasse diesen hier.

```
constructor(obj, anim){
  this.gameObject = obj;
  this.animation = anim;
}
```

Der Konstruktor wird immer dann aufgerufen wenn ein Objekt der Klasse mit new Erstellt wird. Wenn man ein Objekt erstellt kann man der Klasse bzw. dem Konstruktor Parameter mitgeben.

Diese Klasse kann danach sehr übersichtlich durch weitere Funktionen ergänzt werden.

```
12    siren = () => {
13      if (this.gameObject.anims.isPlaying) {
14        this.gameObject.anims.stopOnFrame(this.animation.getFrameAt(0));
15      } else {
16        this.gameObject.anims.play("blue", true, 0);
17      }
18    };
```

(Diese Methode Schaltet das Blaulicht des HLF an und aus)

b. User Interaction

Eine weitere interaktive Spielmechanik ist die Mausinteraktion mit Objekten. Dazu muss das Objekt selbst zum Anklicken ermöglicht werden und deshalb der *FireTruck.js* Constructor angepasst werden:

```
constructor(obj, anim) {
  this.gameObject = obj;
  this.animation = anim;

  this.gameObject.setInteractive();
  this.gameObject.on("clicked", this.onClick, this);
}
```

Die game.js wird dann um folgenden Befehl erweitert, um die Klickaktion auszuführen:

```
function create() {
  this.input.on("gameobjectup", (pointer, obj) => {
    obj.emit("clicked", obj);
  });
```