Overlays

Autoren: Luisa Kurth, Dennis Loska, 06.08.2018

www.overlays.dennisloska.com

Gruppe

Betreuer:

Prof. Dr. Kai-Uwe Barthel

Mitglieder:

Dennis Loska,
Luisa Kurth

Aufgabenstellung

Erstellung eines Spiels (browserbasiert oder als App) zum Trainieren der Wahrnehmung von Farbüberlagerungen. Das Spiel braucht eine gute Spielmechanik, ein Bewertungssystem und eine angemessene Steigerung des Schwierigkeitsgrads. Die Bildüberlagerung soll nachvollziehbar sein und es sollen gut geeignete Bilder (fotorealistisch oder selbst generiert) benutzt werden.

Zeit

Gesamtes Sommersemester 2018 (April bis Juli).

Zur Verfügung gestelltes Material

Das Spiel wurde auf Grundlage eines von Prof. Dr. Barthel geschriebenen Java Programms entwickelt. Dieses Programm beinhaltete die Matrixmultiplikationen zur Überlagerung der jeweiligen Basisbilder zu einem Zielbild, mitsamt der Generierung einer zufälligen binären Matrix (mit der Dimension von "numPics"x"numPics" und "numOnes" vielen 1 pro Zeile) und der dazugehörigen Inversen.

Verwendete Technologien

- Node.js
- Bootstrap
- SASS
- jQuery
- Canvas-API
- Grunt

Installation & Konfiguration

Lokale Entwicklungsumgebung

Zunächst muss Node.js installiert werden. Es sollte hierfür die neuste *LTS* Variante ab version **8.11.1** installiert werden (https://nodejs.org/en/). Nachdem Node installiert wurde sollte man in den *Projektordner* Overlays gehen:

```
cd ./Overlays
```

Als nächstes muss in den *develop* Branch gewechselt werden:

```
git checkout develop
```

Anschließend müssen die Dependencies, also abhängige Node-Module, welche benötigt werden installiert bzw. runtergeladen werden:

```
npm install
```

Nachdem die Dependencies erfolgreich installiert wurden, sollte man den lokalen Server starten können:

```
node index.js
```

Es sollte folgende Meldung im Terminal erscheinen:

```
Node app is running at localhost:5001
```

Wenn neue Features implementiert wurden, empfiehlt es sich pro Feature einen Commit zu machen und Änderungen erst zu pushen, wenn diese auch vollständig sind bzw. funktionieren. Dies wird mit folgendem Befehl realisiert:

```
git push origin develop
```

Grunt - Taskrunner

Wenn das Repository geklont wurde, können wie oben beschrieben die Dependencies, wozu auch Grunt gehört, installiert werden.

```
npm install
```

Nachdem dies erfolgt ist, kann die Grunt CLI installiert werden, um die Grunt-Befehle ausführen zu können. Damit die Bildkompressions-Task funktioniert, müssen zudem noch mit *--save dev* zwei spezielle Dependencies für die imagemin Task installiert werden. Das alles kann mit den folgenden befehlen realisiert werden:

```
npm install -g grunt-cli
npm install --save-dev grunt-contrib-imagemin
npm install --save-dev imagemin-mozjpeg
npm install grunt
```

Falls als OS **Linux** verwendet wird, muss noch libpng16 installiert werden, wenn es im besagten OS nicht bereits schon installiert wurde, damit die Bildkompression funktioniert:

```
sudo apt-get install libpng16-16 //if you don't have it
sudo ldconfig
```

Jetzt kann getestet werden, ob Grunt erfolgreich installiert wurde, indem der Befehl **grunt** in ./Overlays ausgeführt wird.

```
grunt
```

Um das komplette Projekt einschließlich von komprimierten Bildern zu bauen muss folgende Task ausgeführt werden:

```
grunt build
```

Umsetzung des Projektes

- Java Programm komplett in JavaScript umschreiben
- Aufsetzen eines Servers
- Implementierung einer binären User-Matrix wUser, die bei jedem Klick aktualisiert wird und die vom User ausgewählten Basisbilder überlagert
- Implementierung eines Scores (50% Zeit und 50% Klickanzahl), einer Zeitangabe und eines Klickzählers
- Implementierung einer Validierung der Userauswahl (vergleiche User-Matrix mit Lösungsmatrix)
- Implementierung einer ImageGenerator Klasse, die zufällige Bilder mit verschiedenen Formen (Rechteck, Kreis oder Dreieck) mit zufälligen Positionen und Farben malt
- Implementierung eines Seed Farb-Generators mit 3 Tests zur Auswahl von "optimalen" Farben zur Farbüberlagerung (testet Farben auf Abstand zueinander, testet Farben auf Abstand zu 128 / grau, testet Farben auf Grenzbereich nach der Überlagerung)
- Speicherung der getesteten und gut geeigneten Farben in Arrays, aus denen später zufällige Sets ausgewählt werden (Seed und Tests wurden danach rausgenommen)
- Optimierung der ImageGenerator Klasse durch flexible Anpassung an das jeweilige Level: Pro Level kann festgelegt werden, ob die generierten Bilder in Graustufen oder Farben gemalt werden sollen, ob sie eine ähnliche Position haben sollen und ob es ein "leeres" Feld geben soll, in dem keine Form und keine Farbe gemalt wird (einstellbar in der Level.js Klasse)
- Entwurf einer Strategie, die einen steigenden Schwierigkeitsgrad ermöglicht (erst farbig generierte Bilder, dann Graustufenbilder, dann fotorealistische Bilder und mit zunehmendem Schwierigkeitsgrad doGenerate auf true setzen)
- Nutzung von fotorealistischen Bildern (Image.js Klasse) und selbst generierten Bildern (ImageGenerator.js Klasse) in Abhängigkeit des jeweiligen Levels (ebenfalls in Level.js festgelegt)
- Implementierung eines FailedMenu oder Menu, falls ein Level [nicht] geschafft wurde, dass es ermöglicht, in das nächste Level überzugehen, wober die levelNumber erhöht wird und die neuen Parameter für das jeweilige Level aus der Level Klasse geladen werden
- Implementierung einer updateOnClick() Methode in der GameEngine Klasse, die bei jedem Klick auf einen Glasstein aufgerufen wird (erneuert Matrix wUser, berechnet das aktuelle "Zielbild" der Users ausgehend von den angeklickten Basisbildern, prüft ob eine Zeile richtig kombiniert ist, prüft ob alle Zeilen richtig kombiniert sind, aktualisiert das Validierungssymbol der Lampe, zeigt je nach Anzahl richtiger Kombinationen die Punktzahl und das successful / failed Menu an)
- Entwurf eines eigenen Designs (Entscheidung zwischen 2D und 3D)
- Sammlung an sinnvollen Hintergrundbildern (die möglichst einen logischen "Themenbereich" bezogenen Verlauf haben und in einer bestimmten Reihenfolge angeordnet sind) und Rahmen für die einzelnen Bilder anlegen
- Matrixfelder in GUI durch Glassteine ersetzt
- Glassteine auf Klick des Users in einem bestimmten Winkel rotieren lassen
- Lichtstrahlen aus den einzelnen Basisbildern vertikal scheinen lassen (responsive angepasst)
- Lichtstrahlen umleiten, falls ein Glasstein gekippt wird oder wieder in die Ausgangsposition zurückversetzt wird (responsive angepasst)
- Erstellung eines Menus zur Anzeige der Punktzahl und der Zeit, je nachdem ob das Level geschafft wurde oder nicht
- Implementierung einer Dynamit Zeitanzeige, mit sich dynamisch verschiebender Zündschnur, die sich durch Zeit und Klicks bewegt
- Berechnung und Anzeige eines gesamten Scores ("total score")
- Erstellung einer zusätzlichen Infoseite, die Links zum Tech-Stack und den Credits beinhaltet, sowie eine kurze Erklärung des Spiels
- Raussuchen und testen passender schöner Bilder zur Überlagerung (falls keine generierten Bilder von ImageGenerator genommen werden, sondern Bilder aus dem Ordner geladen werden sollen), dessen Farben sich eignen und nicht den Grenzwert (0-255) überschreiten
- Bilder komprimieren, um die Ladezeit zu verkürzen

Klassen

Game.js

- startet Spiel bei Level 0
- erstellt Instanz von GameEngine Klasse
- ruft loadGameGUI() Methode auf

GameEngine.js

- verwaltet die Berechnung der Bildüberlagerung (Basis- und Target images)
- verwaltet Lösungsmatrix und User-Matrix (und somit auch die Überlagerung der des Users ausgewählten Basisbilder)
- erstellt Instanz von Level Klasse durch die loadLevel() Methode und holt benötigte Parameter
- erstellt Instanz von Images Klasse um die benötigten Bilder zu laden
- aktualisiert und validiert die Userauswahl nach jedem Klick mit der updateOnClick() Methode und prüft, ob das Level fertiggestellt wurde
- zeigt die Überlagerung der dynamischen Userauswahl mit drawUserImage() an, nachdem sie mit calculateUserImage() berechnet wurde
- stoppt Zeit und berechnet den Score / die Sterne des Users
- ruft in Abhängigkeit der Punktezahl des Users nach Beenden des Levels showFailedMenu() oder showMenu() auf

ImageGenerator.js

- generiert eigene Bilder mithilfe der Canvas API und dem Uint8ClampedArray (wichtig: hier werden RGBA in aufeinanderfolgenden Indexen gespeichert, also: [0] = r, [1] = g, [2] = b, [3] = a, [4] = r, ... etc.)
- Hintergrund wird mit einem graylevel 128 gefüllt
- verschiedene Formen (an ähnlichen oder unterschiedlichen Positionen) können mit verschiedenen Farben durch die addShapes() Methode gemalt werden
- die zur Verfügung stehenden Farben für die gemalten Formen befinden sich in einem colorsForImages Array und werden durch die getTestedColors() Methode geladen, welche alle möglichen getesteten Farbkombinationen enthält und ein random Set in Farben oder in Graustufen zurückgibt

Images.js

- Fall 1: lädt Bilder aus Ordnern und übergibt sie GameEngine.js zur Überlagerung

- Fall 2: erstellt Instanz von ImageGenerator Klasse um zufällig generierte Bilder zur Überlagerung zu übergeben
- im 2. Fall wird bei der Speicherung des Arrays der zu übergebenden Bilder entschieden, ob die Bilder ein "empty" Bild haben soll und ob die Formen an ähnlichen oder verschiedenen Positionen gemalt werden sollen (kann in Level.js pro Level festgelegt werden), außerdem wird vermittelt, ob es sich um ein Graustufen oder um ein farbiges Bild handeln soll

InverseMatrix.js

- berechnet die Inverse einer Matrix

Level.js

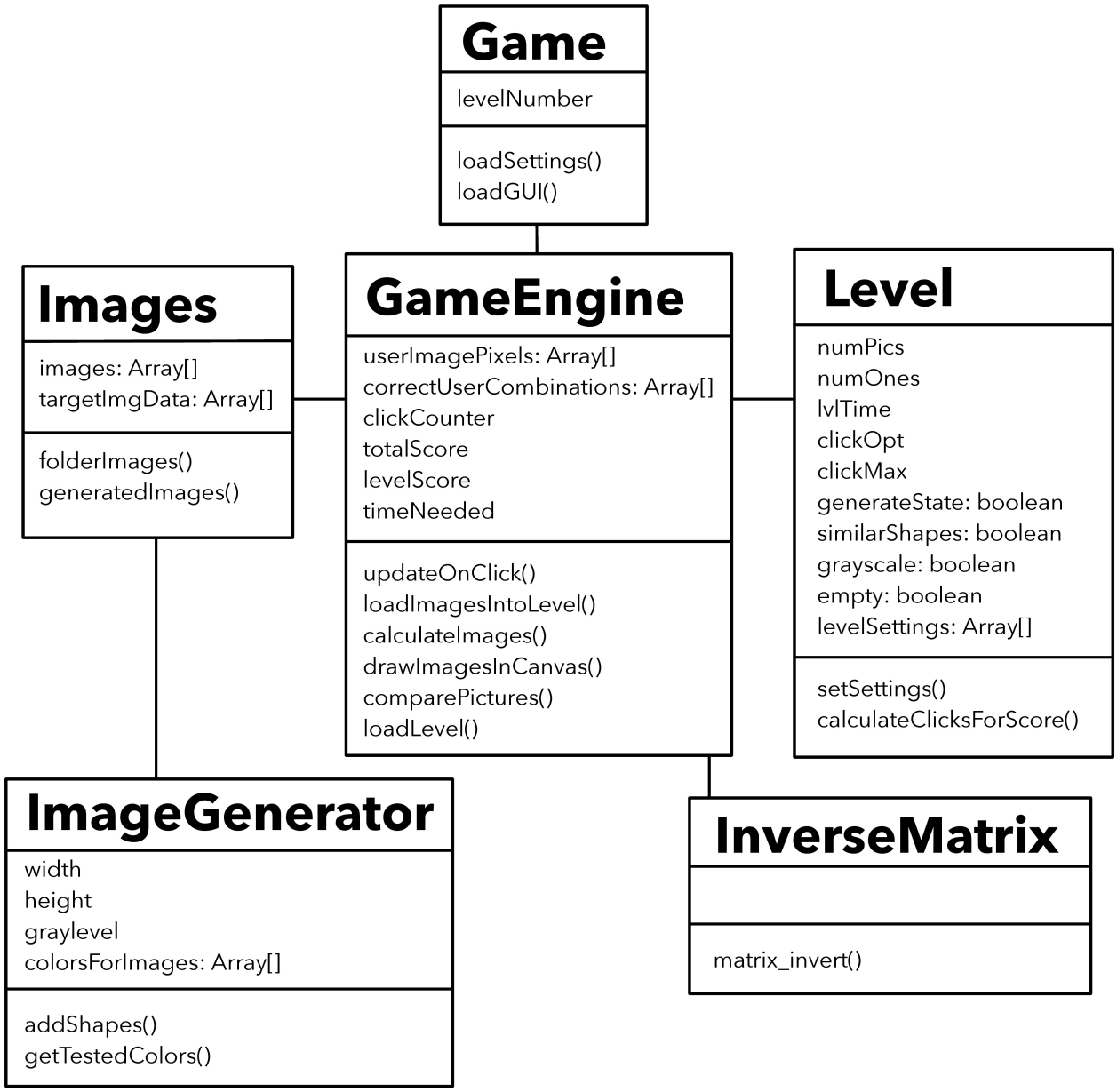
- legt Vielzahl an Parametern für ein bestimmtes Level an, die sich alle flexibel ändern lassen: [numPics, numOnes, doGenerate, gray, empty, similarShapes, folderImage]
- numPics gibt die Anzahl der (Basis- und Target-) Bilder und somit die Dimension der Matrix an (numPics x numPics)
- numOnes gibt die Anzahl der Bilder an, die man für eine richtige Kombination auswählen muss (also die Anzahl an 1en pro Zeile in der Lösungsmatrix)
- doGenerate (boolean) bestimmt, ob die geladenen Bilder als Basis oder Target images verwendet werden sollen
- gray (boolean) gibt an, ob die generierte Bilder in Graustufen erscheinen sollen oder in Farbe
- empty (boolean) gibt an, ob ein Bild leer sein soll (heißt: in einem der generierten Bilder ist nicht als grau zu sehen)
- similarShapes (boolean) legt fest, ob Formen der generierten Bilder an sehr ähnlichen Positionen gemalt werden sollen oder zufällig verschieden
- folderImage (boolean) legt fest, ob in dem Level generierte Bilder verwendet werden sollen (ImageGenerator.js) oder statische Bilder aus dem Ordner geladen werden sollen
- die Methode calculateClicksForScore() berechnet das Optimum und das Maximum an Klicks für das jeweilige Level

View.js

- keine Klasse an sich, sondern Sammlung von GUI-Funktionen
- lädt das gesamte Gerüst des Spiels
- zeigt den Score und die Zeit an
- updateFuseBar() Methode stellt das Dynamit mit der Zündschnur dar und animiert die Bewegung / Verschiebung

Klassendiagramm

Anmerkung: Das Diagramm ist nicht komplett vollständig, sondern zeigt nur die wichtigsten Methoden auf und gibt einen Überblick darüber, wie die Klassen miteinander verbunden sind.



Funktion des Spiels

Das Spiel zeigt dem User in jedem Level die zu überlagernden Basisbilder (oben, horizontal) und die zu erreichenden Zielbilder (ganz rechts, vertikal) an. Von jedem Basisbild geht ein Lichtstrahl aus, der vertikal nach unten leuchtet, solange man ihn nicht umleitet. In der Mitte befindet sich eine Matrix auf Glassteinen, die ihre Position verändern, wenn man sie anklickt und dadurch den vertikalen Lichtstrahl nach rechts im 90 Grad Winkel umleiten.

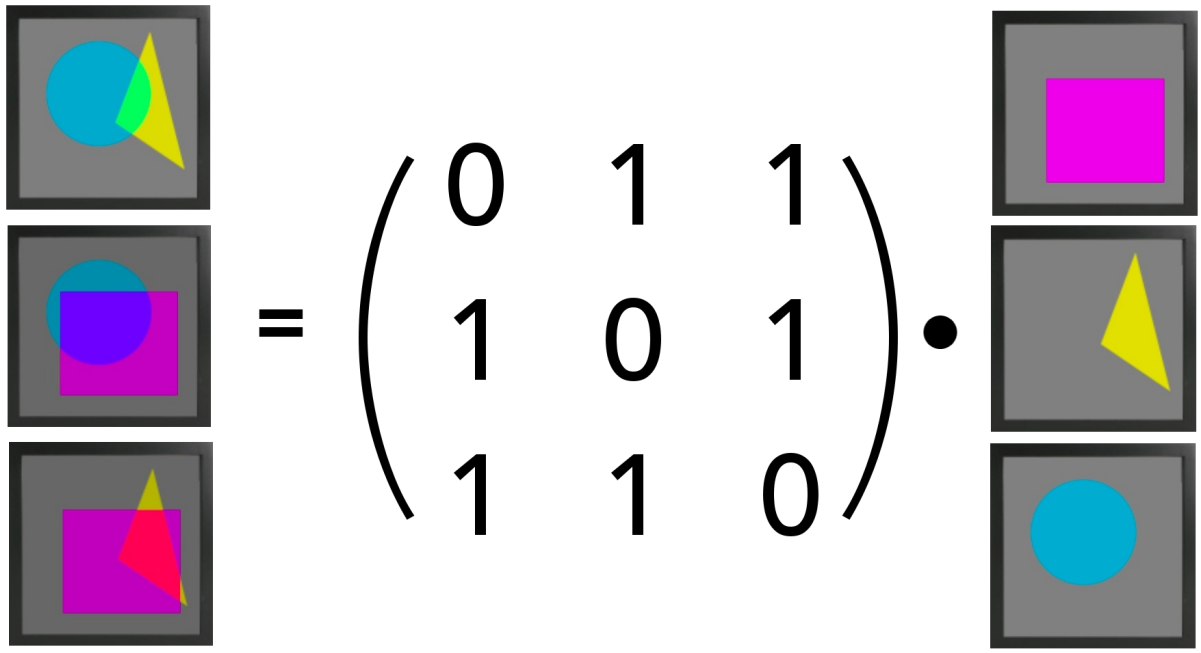
Wenn ein Lichtstrahl umgeleitet wird, dann bewirkt das, dass das Basisbild, das den Strahl aussendet, rechts in der vertikalen Spalte neben den Zielbildern angezeigt wird. Wählt man mehrere Bilder in einer Reihe aus – also: werden mehrere Glassteine in einer Reihe gekippt und leiten die Strahlen um – dann werden diese Bilder nach den Regeln der additiven Farbmischung überlagert. Diese Bilder werden als User-Images bezeichnet (eins pro Zeile).

Wenn ein User-Image in einer Zeile gleich dem Zielbild der selben Zeile ist, dann wurde eine korrekte Kombination gefunden und die Lampe auf der rechten Seite wird grün. Wenn alle Lampen grün leuchten, der Nutzer also alle Bilder richtig zu den vorgegebenen Zielbildern kombiniert hat, dann ist das Level geschafft. Es wird ein Score ausgegeben, der sich aus der benötigten Zeit (50%) und der benötigten Anzahl an Klicks (50%) errechnet, sowie eine Anzeige von 0-3 Sternen für das Level (die sich aus dem Score ergeben).

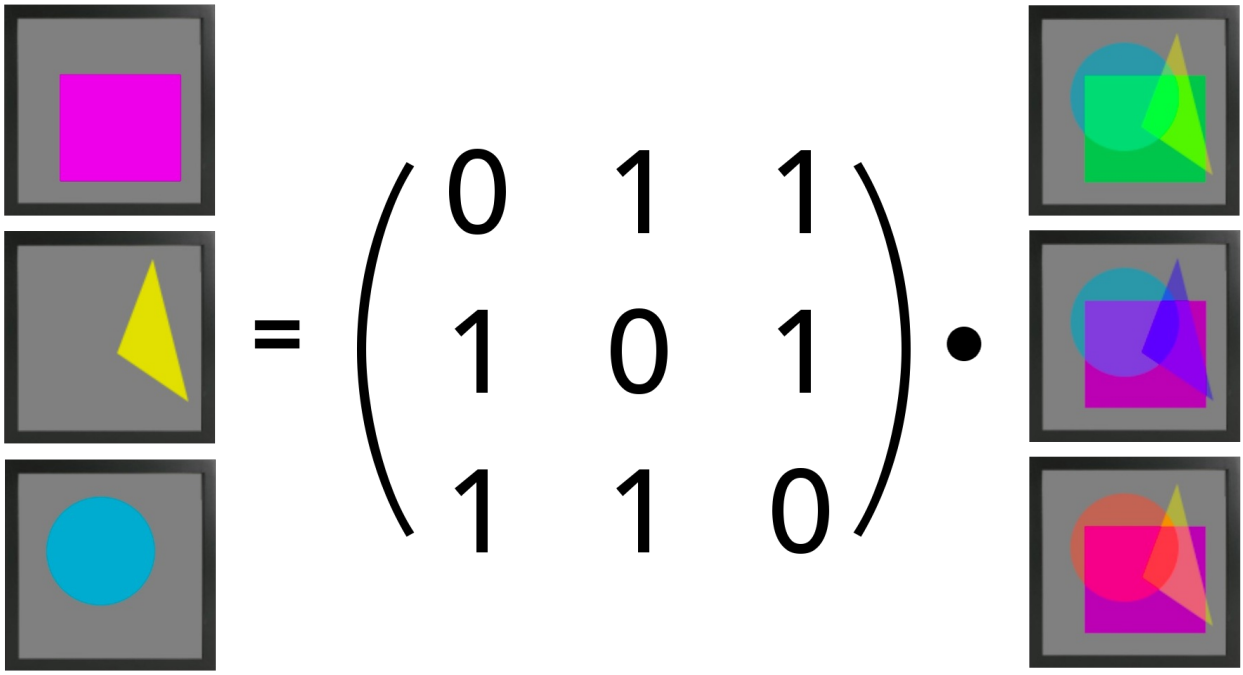
Die Glassteine können beliebig oft angeklickt werden und die Reihenfolge der Auswahl der Basisbilder zur Überlagerung ist nicht wichtig / ausschlaggebend für das entstehende Bild. Ist die Zündschnur (= die dynamische Anzeige der zu verbleibenden Zeit / verbleibenden Klicks) unten im Bild abgelaufen, bevor der Nutzer alle Zeilen richtig kombiniert hat, ist das Level verloren. Diese verkürzt sich durch abgelaufene Zeit und mit jedem Klick auf einen Glasstein.

Spielmodi

doGenerate = false



doGenerate = true



Erweiterungsmöglichkeiten

- weitere Anpassungen und Optimierungen des Designs und zusätzliche Animationen (z.B. Explosion des Dynamits nach Ablaufen der Zeit, Rahmen um Bilder auch von Level zu Level verändern). Hierfür wurde bereits eine JS-Datei mit dem Namen *animations.js* im Ordner *js/utils* angelegt
- Anbindung an API zur Verwendung eigener / anderer Bilder (Instagram, Flickr etc.)
- Anbindung einer Datenbank zur Speicherung des Punktestands (Score und Name des Users)
- Möglichkeit einzelne Level zu wiederholen und nicht immer von vorne anfangen zu müssen
- Level "freischalten" mit Punkten oder Sternen
- weitere ImageSets erstellen oder heraussuchen
- Einstellungen und gewisse Variablen in eine Konfigurationsdatei auslagern (z.B. die Konfiguration der einzelnen Level)