

Technische Dokumentation

Projekt: Fotomosaik-Generator

Mitglieder: Johannes Benzing

Onur Sahin

Sascha Mander

Verwendete Technologien: HTML5

CSS

Javascript (mit jQuery-Bibliothek)

Ordnerstruktur: css - Eigene Stylesheet Dateien

images - Bilder (z.B.:Favicon)

js - Eigene (Java-)Skripte

plugins - Extern verwendete Bibliotheken (z.B.: jQuery)

index.html - einzige HTML-Datei

Wichtige Skripte: *benzingLightbox.js*

Lightbox für die Bilder, die als Kacheln verwendet werden sollen.

Erlaubt auch das Löschen und Rotieren dieser Bilder.

canvasZoom.js

Erlaubt Zoomen und verschieben des Bildausschnitts des fertigen Mosaiks.

configAndGlobalsAndConstants.js

Hier werden alle globale Variablen und Konfigurations-Variablen und Konstanten definiert.

dropzonesUpload.js

Die Funktionalität der input-tags und der Drag'n'Drop-Felder wird in durch diese Datei ermöglicht.

exporter.js

Ist für den Download des fertigen Mosaiks in unterschiedlichen Formaten z.b. .png verantwortlich.

mosaic.js

Beinhaltet Funktionen zur Generierung des Mosaiks und eine Vorschaufunktion wie das fertige Mosaik aussehen könnte

index.js

Enthält Logik der input-tags für die Einstellungen wie z.B. Breite und Höhe der Kacheln.

Verwendete Algorithmus zur Mosaic Erstellung:

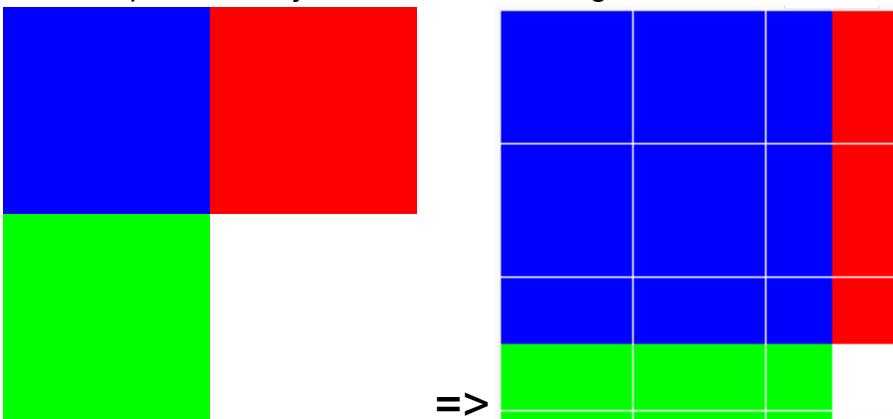
1. Von Bildern, die als Kachel verwendet werden sollen, durchschnittlichen RGB-Farbwerte ermitteln
z.B: Aus dem "Blatt-Bild" RGB(10,230,15) ausgelesen...



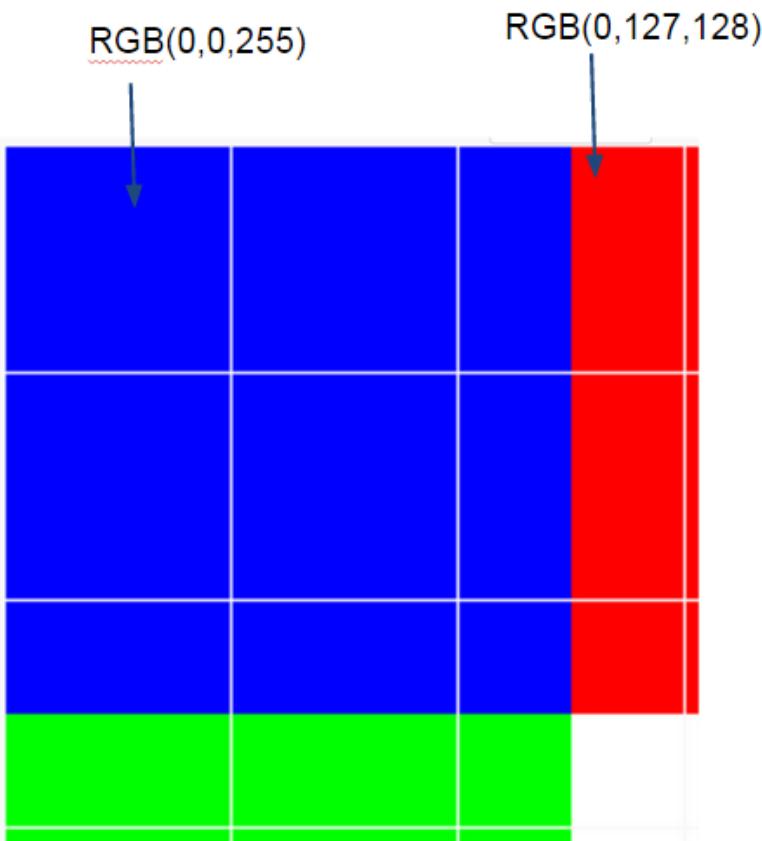
und aus "Himbeeren"-Bild: RGB(240,5,10)



2. Das Haupt Bild wird jetzt in Kacheln zerlegt

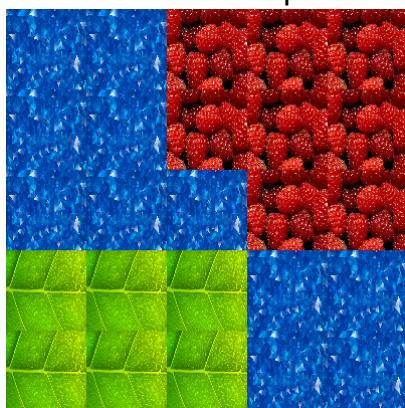


3. Farbwerte der Kacheln des Haupt Bildes auslesen. Dabei werden alle Reihen und Spalten des Haupt Bildes in einer verschachtelte For-schleife durchlaufen.



4. Passendes Bild finden(näheres: siehe Datenstruktur) zur Kachel finden und in Mosaik einfügen:

So würde das fertige Beispiel aussehen, wenn man nur 3 Kachel-Bilder nutzen würde und das Hauptbild in nur eine geringe Anzahl an Kacheln zerlegen würde.



Verwendete Datenstruktur (kd-tree.js):

Die Software nutzt einen 3-dimensionalen kd-tree wegen der schnellen Nearest-Neighbour-Suche bei einem 3-dimensionalen Suchschlüssel(die 3 RGB-Farbwerke)

Ein kd tree ist ein k-dimensionaler Binärbaum, bei dem jeder Knoten ein k-dimensionaler Punkt ist.

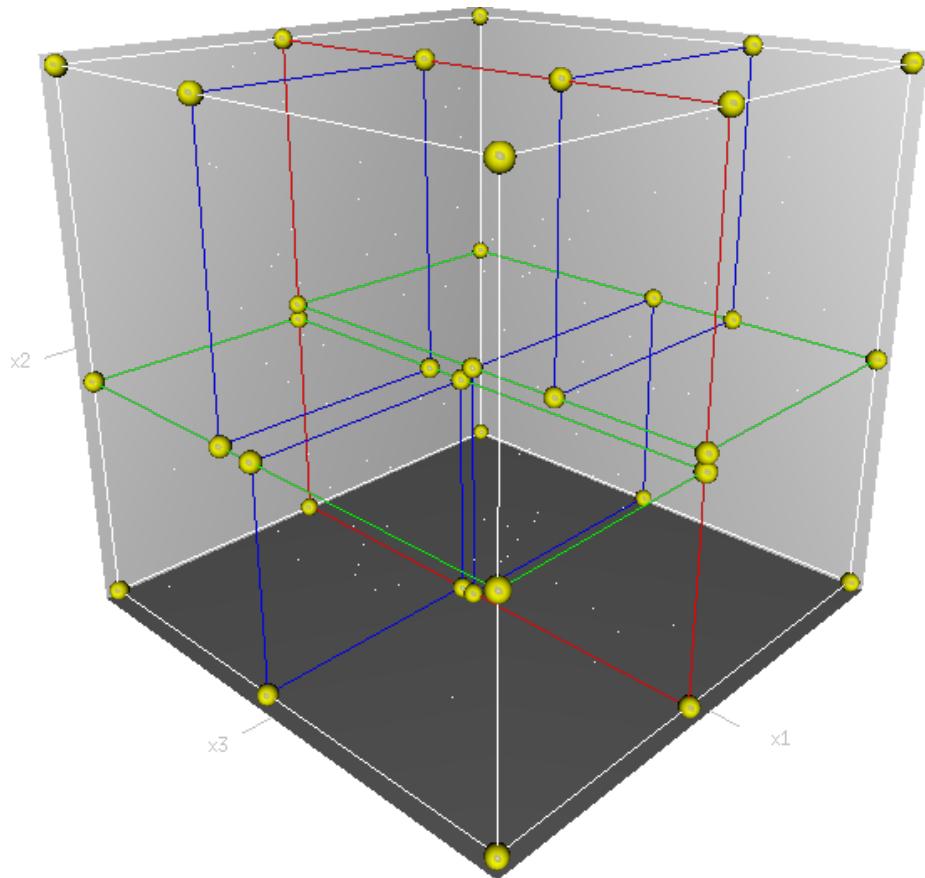


Abb. eines 3-dimensionalen-Trees

Algorithmus Nearest Neighbour search bei kd-trees:

- Beginnend beim Startknoten wird im Baum rekursiv nach unten gegangen. Dabei jeweils nach links/rechts je nach Wert der Hyperebene und des RGB-Werts des Punktes, zu dem ein gleicher/ähnlicher Punkt gesucht wird
- Nachdem ein Blattknoten des Baums erreicht wird, wird dieser Knoten als "momentan bester Knoten" gespeichert
- Der Weg vom Blattknoten zum Startknoten wird rückwärts zurückgegangen, bei jedem Knoten wird folgendes überprüft:
 - ob einer der Knoten näher als der "momentan bester Knoten" liegt. Wenn dem so ist, wird der aktuelle Knoten als neuer "momentan bester Knoten" gespeichert.
 - Außer dem wird überprüft ob auf der anderen Seite der aktuellen Hyperebene ein näher liegender Punkt sein könnte:
 - Falls Abstand zwi. "momentan besten Knoten" und dem gesuchten Punkt kleiner als der Abstand "momentan bester Knoten" und aktueller Hyperebene
 - dann weiter mit nächstem Knoten den Baum hoch in Richtung des Startknotens => Punkt kann nicht auf der gesamten anderen Seite der Hyperebene sein
 - Falls Abstand zwi. "momentan bester Knoten" und dem gesuchten Punkt größer als der Abstand "momentan bester Knoten" und aktueller Hyperebene
 - dann muss der Algorithmus, den anderen Zweig des Baums des aktuellen Knotens auch rekursiv auf nähere Punkte untersuchen

Am Ende des Algorithmus hat man den Nearest Neighbour des gesuchten Punktes in der Variable "momentan bester Knoten".

Fazit:

Grundsätzlich lässt sich sagen, dass die Generierung eines Mosaiks funktioniert.

Weil Javascript in nur einem Thread(inklusive des UIs) läuft und Webworker nicht auf DOM-Elemente wie Canvas-Elemente zugreifen kann, muss zum Generieren des Mosaiks die Javascript-Funktion

`setTimeout(function, milliseconds);`
verwendet werden, damit das UI ab und zu Rechenzeit erhält. Würde man ohne
`setTimeout(function, milliseconds);`
arbeiten(also eine sehr lange for-schleife ohne Pause laufen lassen), würde sich der Browser aufhängen.

Leider kann man nur abschätzen wie groß dieser Timeout zwischen den einzelnen Aufrufen sein muss. Deshalb haben wir den Timeout-wert auf einen sehr hohen Wert gesetzt, damit es in allen Browser und auch auf langsamer Hardware funktioniert. Was aber dazu führt, dass es in schnellen Umgebungen länger, dauert als es müsste. Deshalb lässt sich in der Anwendung der Timeout einstellen auf Risiko hin, dass der Browser crashen könnte.

Unser Erfahrung nach ist Chrome mit Abstand am schnellsten beim Generieren, danach folgen Firefox und Opera. Am langsamsten war Internet Explorer

Außerdem kommt es oft zu Abstürzen der Browser, wenn man versucht extrem große Mosaiks als *.jpg zu speichern. Jedoch ließen sie sich als *.png speichern. Es wird vermutet, dass folgende Methode ein browserabhängiges Limit, wie groß der String sein darf, der returnt wird, hat:

`canvas.toDataURL()`