FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Generierung von Saalplanbildern mithilfe von Clustering und Algorithmen zur Erstellung konkaver Hüllen

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Maximilian Deutsch
Matrikelnummer 1327587

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Hsiang-Yun Wu

Wien, 21. Oktober 2018

Maximilian Deutsch	Hsiang-Yun Wu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Generating seating plan images using clustering and concave hull algorithms

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Maximilian Deutsch
Registration Number 1327587

to the Faculty of Informatics

at the TU Wien

Advisor: Hsiang-Yun Wu

Vienna, 21st October, 2018

Maximilian Deutsch

Hsiang-Yun Wu

Erklärung zur Verfassung der Arbeit

Maximilian Deutsch
Ameisgasse 52

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Oktober 2018

Maximilian Deutsch

Kurzfassung

Diese Studie präsentiert ein Verfahren zur Generierung von Saalplänen der Ticket Gretchen app. Die App bietet die Möglichkeit Tickets für Theater und ähnliche Veranstaltungen mithilfe eines interaktiven Saalplanes zu kaufen. Ein Saalplanbild ist eine abstrakte Visualisierung eines Veranstaltungsortes die auf dessen Sitzanordnung basiert. Das Bild soll einen Eindruck der räumlichen Struktur geben, um zu erkennen welche Sitze sich in Reichweite voneinander befinden. Die vorgeschlagene automatisierte Lösung der Generierung dieser Bilder ersetzt den zuvor verwendeten Prozess, die Saalplanbilder manuell zu erstellen. Das Bild besteht aus Polygonen, die Sitzgruppen darstellen, die dem Benutzer zeigen, welche Sitze nahe beieinander liegen und welche voneinander getrennt sind. Die Gruppierung der Sitze ist durch den DBSCAN Clustering Algorithmus mit der Vewendung der 2D Position, Sektor- und Logeninformation realisiert. Für die Berechnung der Polygone werden zwei Algorithmen zur Erstellung von konkaven Hüllen verglichen.

Abstract

This study presents a process of generating seating plan images for the Ticket Gretchen app. The app offers the ability to buy tickets for theaters and similar venues by using an interactive seating plan. A seating plan image is a venue's abstract visualization defined by the seating layout of a performance. It should give an impression of the spatial structure to see which seats are in reach of each other. The proposed automated solution of generating these images replaces the previously used process of creating the seating plan images manually. The image is made up of polygons representing seat groups that show the user which seats are near each other and which are separated from each other. The grouping of seats is done with the DBSCAN clustering algorithm using the seats' 2D position, sector and box information. For the computation of the polygons two concave hull algorithms are compared.

Contents

CHAPTER 1

Introduction

This project presents a process of creating seating plan images for theater plays and similar events. When buying tickets for such an event it is advantageous to have a visual representation of the venue to know where your seat is going to be located. This visualization should give the visitor enough information about the venue's structure to make a decision where they wanted to be seated. Therefore the seating plan is a crucial part of buying tickets for an event.

Ticket Gretchen[tic] is an app to buy tickets for events where users can select their seats with an interactive seating plan shown in Figure 1.1. The seating plan shows available seats for an event at a a specific time, which will be referred to as a *performance.* The location an event takes place is called a *venue.* The colored dots represent the seats the user can book with the color representing the price category. The small grey crosses display unavailable seats. All seats are placed on white shapes forming a so called *seat group.* Seat groups represent a collection of seats that should be visually combined, to depict which seats are in reach of each other. For example in a theater you can find boxes that are spatially divided from each other. In Figure 1.1 the horseshoe shaped seats around the center of the image are individual boxes, each represented by a white shape. The stage can be found at the bottom of the image displayed as a rectangle. The seating plan image only consists of the seat groups and the stage. The seats are drawn on top of the image by the app.

Each venue, event and even performance can have a different seating plan. For example there is a standard seat layout for a venue, for which a seating plan image is required. Some events may change the seat arrangement by omitting sections, whereby an own image needs to be created. Also this is possible for a specific performance of an event. Additionally when new venues and events are included in the app, new seating plan images have to be created. For Ticket Gretchen there is an ongoing need of new seating plan images.

Up until now the seating plan images were created manually. For this, the seats for a performance are rendered onto a blank image using the seats' X- and Y-positions. This image is then opened in a vector drawing program to draw the seat group polygons. The decision which seats should be grouped is based on experience and knowledge of the venue. For example, the creator looks up a photograph of the place in order to know that some seats form a box and therefore will be grouped together. After all seats are contained in a shape the stage is drawn at its appropriate location and the seats are removed. This is a very cumbersome process that takes a lot of time.

The goal is to simplify the process of creating new seating plan images. This is done by having an automated solution which is fully integrated into the current system. The generated seating plan images should represent the seating layout as good as possible. The seats should be grouped based on their location and any other available information that separates them from another. Each seat needs to be included in a seating group and no seat groups should overlap each other. The stage should be placed at an appropriate location, which the user can change afterwards.

The developed solution manages to generate similar seating plan images compared to the manual created ones. Creating seating plan images is now simpler, because the user only needs to specify some parameters from which the image is automatically generated. This reduces the time it takes to introduce new venues to the app. The method creates seat groups by clustering the seats based on position, sector and box information using DBSCAN[EKS+96]. Subsequently polygons for the seat groups are computed using a concave hull algorithm. For this two methods have been compared. The stage is positioned automatically, but can be modified in the user interface.

First related work will be discussed in chapter 2. In chapter 3 the proposed method of generating seating plan images is presented. Any implementation details and the user interface are explained in chapter 4. The solution is evaluated in chapter 5 where different results are presented and limitations discussed. Finally in chapter 6 a conclusion of the project is given and potential improvements are discussed.

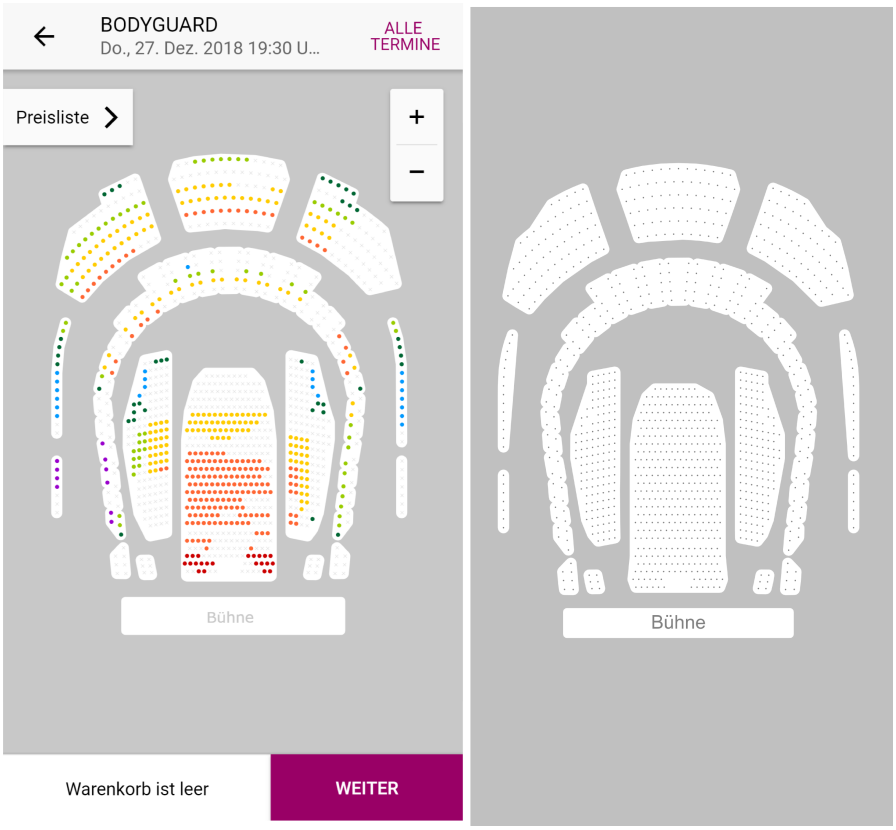


Figure 1.1: On the left side is the app's interactive seating plan where available seats are drawn on top of the manually created seating plan image. On the right side the corresponding generated seating plan image can be seen.

CHAPTER 2

Related Work

The task of this thesis is very specific and no comparing products can be found that offer the required functionality. The following software solutions for seating plans in the ticketing domain have been found:

The product *seats.io* [seab] offers the ability to create and embed interactive floor plans to sell tickets. Their solution allows one to build a seating plan manually with an editor. It is then accessed via an API and embedded with JavaScript. With the editor it is not possible to automatically generate a seating plan from some input data.

Social Tables [soc] is an event management software to organize event sales collaboratively. It includes the product to create diagrams and seating charts manually with the ability to convert them to 3D representations. But there is no option to create an image automatically from existing seating data.

With the ticketing solution *SeatAdvisor* [seaa] it is possible to create so called *SeatMaps* for venues. Again it is not possible to generate seating plan images automatically, but only manually.

The patent application from Stanley H. Kim [Kim14] describes a system for generating a venue's seating chart dynamically. To do so the user may input photos or drawings of the location, which then are used with object recognition software to identify individual seats. Alternatively the user may provide the venue's dimensional information by inputting the number of rows and seats for each section. The patent only describes the system in an abstract way and does not provide any details for a possible implementation.

2.1 Clustering

Clustering is grouping objects together so that elements within a cluster are more similar to another than to ones in different clusters. It is exploring data with no or little

information in advance. Clustering is a well known field of study with approaches suitable for different applications.

One of the most popular clustering algorithms[XW09] is the k-means algorithm[M+67]. It partitions a data set iteratively into k clusters by assigning the points to their nearest cluster. The number of clusters and their initial centers need to be determined beforehand. For a seating plan image the number of seat groups is not known in advance. Also the clusters of the k-means algorithm tend to form circular shapes which is not desired for the seating plan image.

Hierarchical clustering techniques create a hierarchy of results with different levels of proximity. Agglomerative hierarchical clustering[MC12] iteratively merges elements based on their similarity. Each merging step produces a clustering result from which a desired one is selected using a similarity threshold. Because the algorithm has a computation complexity of $O(n^2)$ and there is no need for the hierarchical structure, it was decided not to use this technique.

2.2 Polygon computation

The clustered seat groups need to be represented as polygons. These polygons can be created using the Graham Scan[Jar73] which calculates the convex hull of a set of points. However in this project the seat groups need to be represented as concave shapes.

The k-nearest neighbours approach for concave hulls[MS07] sequentially creates the boundary of a set of points. Going from a starting point the algorithm selects the next best point from its k nearest neighbours by looking at the angles the edges will create. Simply put the algorithm tries to make the largest right-hand turn compared to the previous edge. The value k is an input parameter and controls the smoothness of the polygon. Duckham's concave hull algorithm[DKWG08] used in this thesis also uses an input parameter controlling the smoothness. This algorithm was used because of computation complexity of $O(n\log n)$ is better than the k-nearest neighbours approach with computation complexity of $O(n^3)$.

CHAPTER 3

Seating Plan Image Creation

The developed process for creating seating plan images can be examined in Figure 3.1. It includes actions that are performed by the user (input of data) and actions that are performed automatically (image generation itself). The creation of a seating plan image is based on the seating information of a single performance. A performance is an event at a concrete time which determines the layout of the seats. Additionally the user inputs the parameter values that are needed for the subsequent steps and initiates the automatic procedure. First the seating data for the selected performance is retrieved from an external system. The seats are then clustered into seat groups that represent a unit of seats which are visually combined. Next polygons are created for these seat groups and an appropriate position of the stage is calculated. Finally the SVG image is created from the polygons including the stage. The image is then displayed to the user who decides if they are satisfied with the result. If not, the automatic process is redone with different parameters. If the image is satisfying it is then possible to manually manipulate the stage, concluding the seating plan image creation.

3.1 User input

First the user needs to specify the performance for which the seating plan image is created for. This is done by searching for an event and selecting a performance on a specific date. The performance is important, because it defines the layout of the seats. It would be possible that an event has different seat layouts at different times. Also offsets can be specified to shift the seats' positions. Additionally the user inputs the parameter for the clustering, polygon creation and stage calculation.

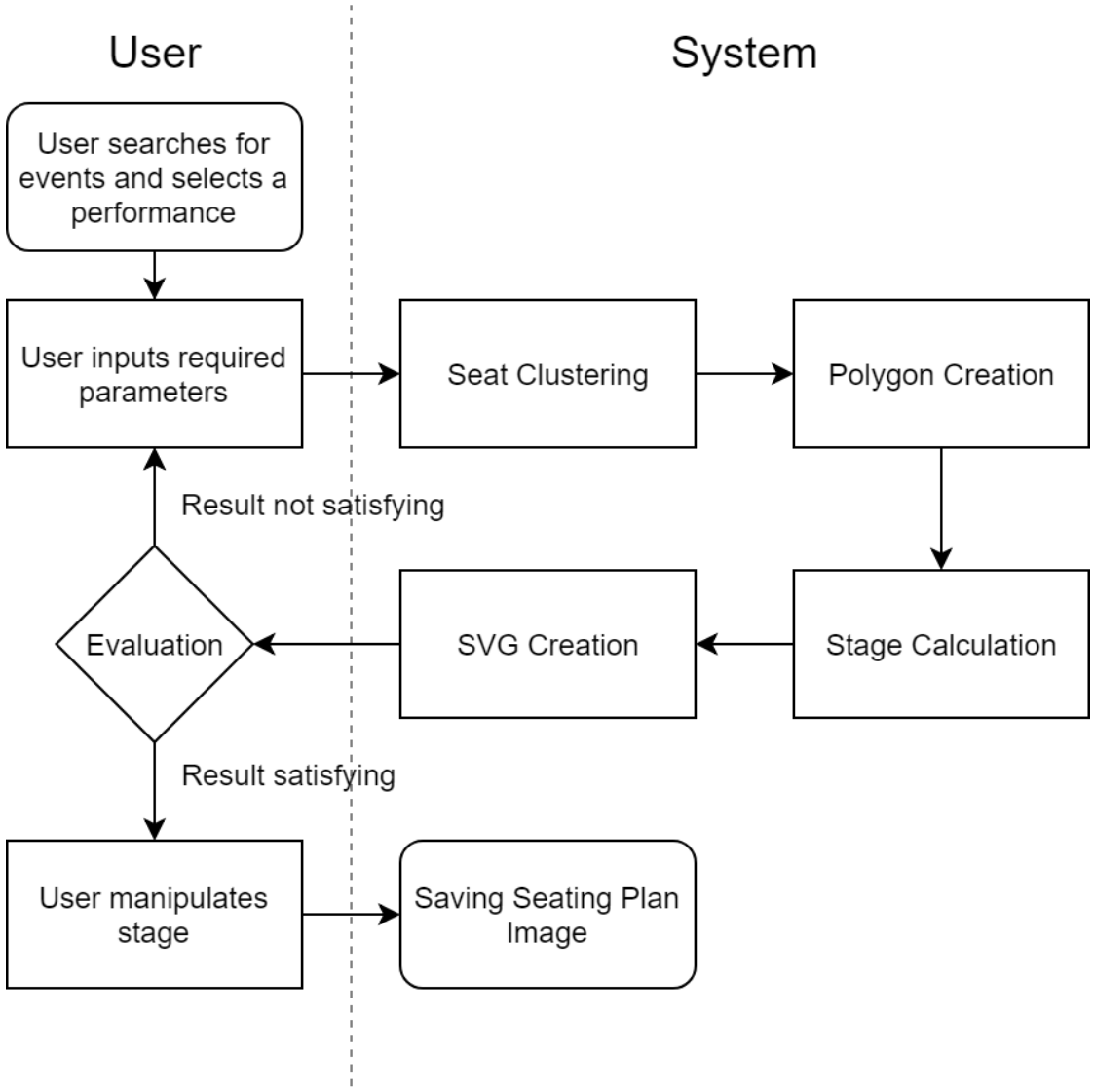


Figure 3.1: Flowchart of the process for creating a seating plan image.

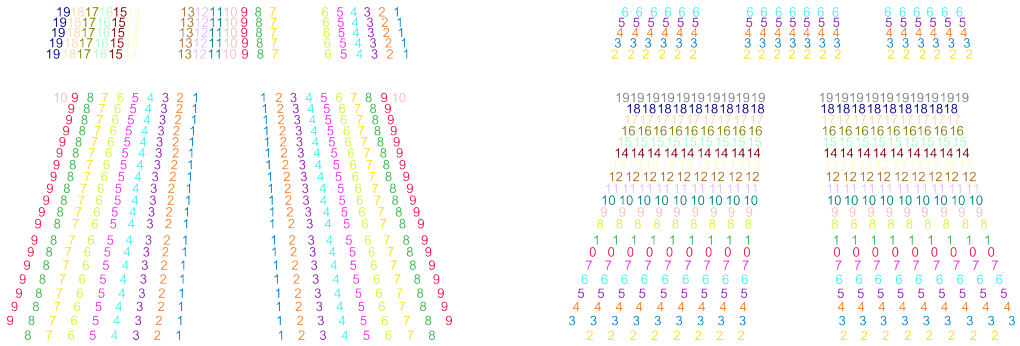
3.2 Seat data

All the data regarding a performance including the seats is retrieved from external sources. Figure 3.2 displays the data of a single seat and a visualization for each attribute. The attribute *seatId* identifies a single seat distinctively. *row* is the name for a group of seats typically aligned in a row. *seatNumber* identifies a seat within a row. *sectorId* identifies a group of seats that usually consists of one ore more rows. *ticketCategoryId* identifies seats in the same ticket category which determines the ticket's price. $posX$ and $posY$ make up the 2D coordinates of a seat.

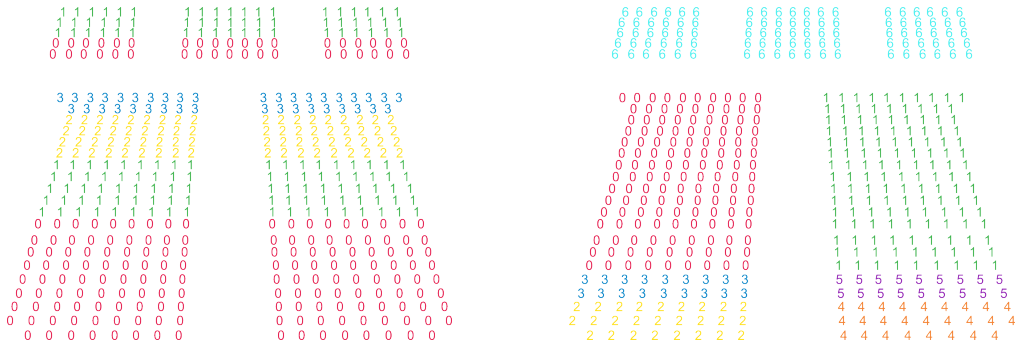
The 2D positions of the seats don't originate from a top down view of the venue. They are projected in a way that no seats would overlap. Seats on different floors that are right above each other were shifted in a way that the overall structure is still recognized.

Feature Name	Data Type	Example
seatId	String	511636_1_5
row	String	Proszeniumsloge links 1
seatNumber	String	1
sectorId	Number	4572
ticketCategoryId	Number	26985
posX	Number	729.81
posY	Number	1168.29

((a)) A seat's features with their data types and examples.



((b)) *seatNumber* ((c)) *row*



((d)) *ticketCategory* ((e)) *sector*

Figure 3.2: Subfigure 3.2(b) shows the seat's actual value. Subfigures 3.2(c), 3.2(d) and 3.2(b) show the values mapped to sequential numbers. The color shows seats with the same value.

Boxes are important for the resulting image because they are often separated by walls from other seats. Therefore seats in the same box must be in the same seat group.

The box information is not directly within the data, but can be derived from the row information. For example two seats in the same box have the following row value:

"Loge 8 links 1. Reihe"
"Loge 8 links 2. Reihe"

They belong to the box with the name "Loge 8 links". What is also included is the number of the row within the box which has to be removed first. Assuming the row number is present in a numerical way the row string is cut off before the last number leaving the name of the box. This string can now be used to identify boxes.

3.3 Creating seat groups with clustering

For creating seat groups the clustering algorithm DBSCAN [EKS+96] is used. It is a density based approach and allows the creation of clusters of arbitrary shape with no prior knowledge of the number of clusters. DBSCAN examines the *Eps*-Neighbourhood $N_{\text{Eps}}(p)$ of a point p defined in Equation (3.1). It uses a distance function $d(p, q)$ between two points p and q from a Database D.

$$N_{\text{Eps}}(p) = \{q \in D |\ d(p, q) \leq \text{Eps}\} \tag{3.1}$$

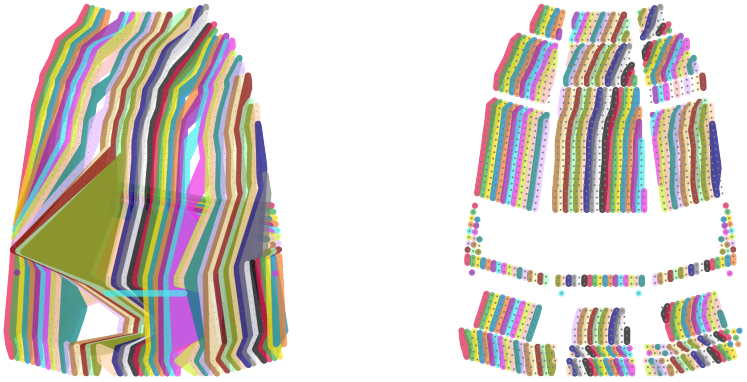
The algorithm distinguishes core and border points which both belong to a cluster. Core points have at least MinPts points in their Eps-Neighbourhood: $|N_{\text{Eps}}(p)| \geq \text{MinPts}$. Border points don't have enough neighbours, but are contained in the neighbourhood of core points. All other points are considered as noise. As every seat needs to belong to a seat group MinPts will always be set to 0. Consequently clusters containing only one seat are possible. Eps is an input parameter that is chosen by the user.

3.3.1 Testing seat attributes for clustering

Figures 3.3, 3.4 and 3.5 show clustering attempts that use the seat number, row and ticket category information. Each color corresponds to a unique attribute value. The figures on the left show the clustering results using only the respective attribute, whereas the right figures show the combination with the position attribute. It can be observed that using the attributes on their own don't produce good results, because their values appear at many locations. Therefore the position is needed so that a cluster does not span across the whole image. Row and seat number create too many clusters. The ticket category combined with the position gives the most promising result shown in Figure 3.5(b). However this is also found to be unsuitable because it separates seats which are not spatially divided.

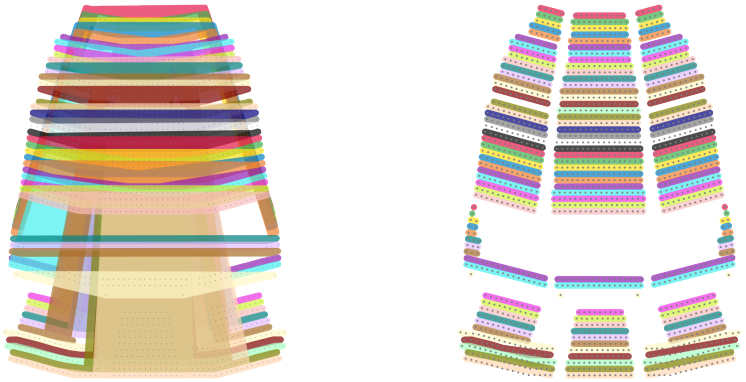
3.3.2 Final distance measurement

For the final distance measure the position, sector and box information is used to determine the distance between seats. The function is notated in Equation (3.2). The



((a)) *seatNumber* ((b)) *seatNumber* and *position*

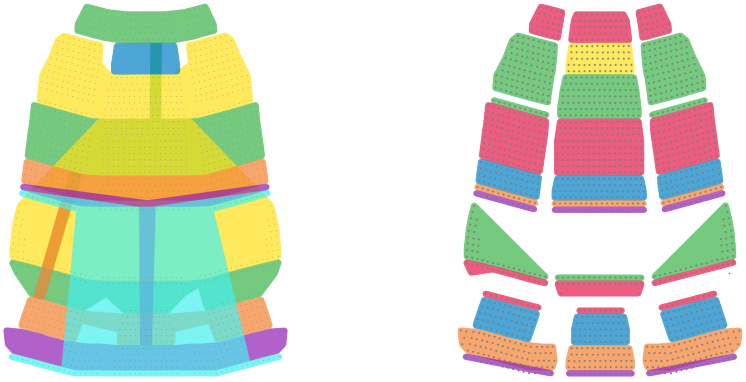
Figure 3.3: Using seatNumber for clustering



((a)) *row* ((b)) *row* and *position*

Figure 3.4: Using row for clustering

ordering of the cases determine its evaluation. The box information is most important. If two seats are in the same box they must definitely end up in the same seat group and therefore get assigned a distance of 0. If only one is in a box or they belong to different boxes they must be separated and therefore the greatest distance possible is assigned. In some cases it is found that clustering sectors together produce good results. Therefore the user has the option to try to group seats based on sectors. In any other cases the proximity between two seats is defined by the euclidean distance. In Figure 3.6 a clustering result using the position only can be seen. Results using different *Eps* values



((a)) *ticketCategoryId* ((b)) *ticketCategoryId* and *position*

Figure 3.5: Using ticketCategoryId for clustering

can be found in section 5.1.

$$d(p,q) = \begin{cases} 0, & \text{if } \text{box}_p = \text{box}_q \\ \infty, & \text{if } \exists \text{box}_p \veebar \exists \text{box}_q \vee \text{box}_p \neq \text{box}_q \\ 0, & \text{if } \text{sector}_p = \text{sector}_q \text{ (optional)} \\ \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2} & \text{otherwise} \end{cases} \tag{3.2}$$

3.4 Computing polygons that represent seat groups and the stage

After creating the seat groups it is now necessary to compute polygons that represent them as good as possible. A group of points could be represented by a well defined unique convex hull, but such a shape may not represent them in the best way possible. Taking a group of points in a "U" shape as an example: The resulting convex shape would more look like an "O" and would not describe the group sufficiently. Therefore it is necessary to use a concave hull, also known as alpha shape, characteristic shape or chi shape as described in [DKWG08]. The problem with finding a concave hull is that there is not one unique solution. A group of points can be described by multiple concave hulls. Two approaches will be compared in the next sections.

3.4.1 Duckham's concave hull algorithm

The first approach for the polygon creation is based on Duckham's algorithm[DKWG08]. It has one input parameter l which controls the smoothness of the resulting polygon.

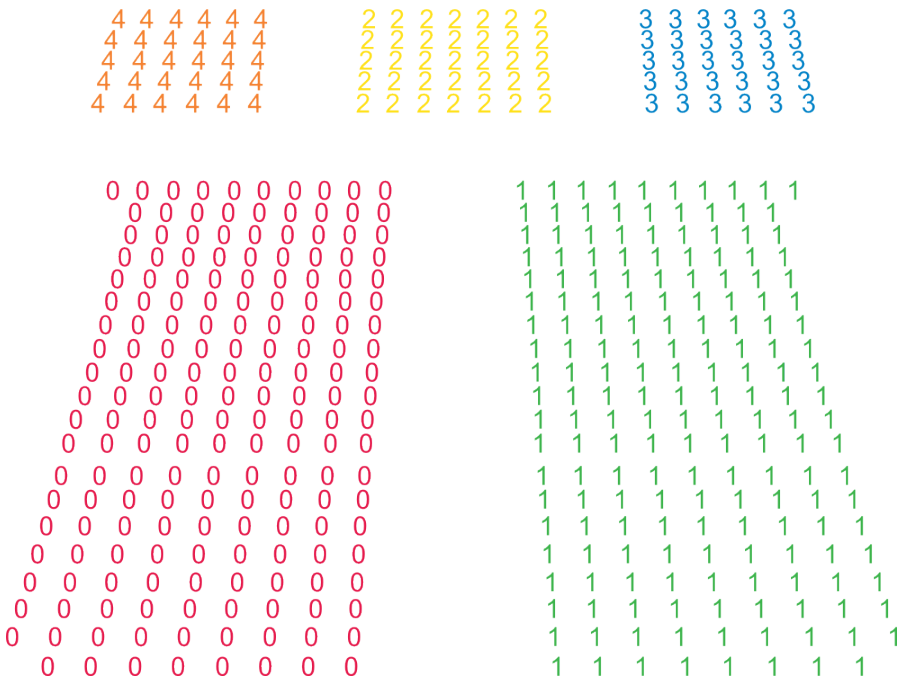


Figure 3.6: Clustering result into seat groups using the seats' position. The number correspond to the seat group's id.

First a Delaunay triangulation of the points is generated. Then the longest boundary edge is removed if it is longer than l and the resulting exterior edges would still form a simple polygon. This is repeated as long as there are edges left to be removed.

The algorithm works on oriented combinatorial maps which is a data structure representing graphs. Such a map is defined by so called *darts*, pairs of darts which form an edge and darts which represent a vertex. Edges and vertices are defined in cyclic notation. Figure 3.7 illustrates a combinatorial map. For example the darts $(2, 3)$ form an edge. The darts $(3, 4, 5, 6, 7)$ form a vertex and are ordered in a anticlockwise manner.

To determine if an edge belongs to the boundary the algorithm checks for a property that is only true for darts of interior edges. Simply explained the algorithm turns from the dart to the left and then to the opposite, repeating this 2 times. If the starting dart has been reached again there must be a triangle to the dart's left. If this is true for both darts the edge they form can only lie inside the graph, concluding which edges belong to the boundary.

An edge will only be removed if the resulting boundary still forms a simple polygon. This is true if the vertex lying on the opposite of the edge belongs to the interior. If it is an boundary vertex a simple polygon cannot be formed anymore.

When seat groups are arranged in a line they cannot be represented as simple polygons.

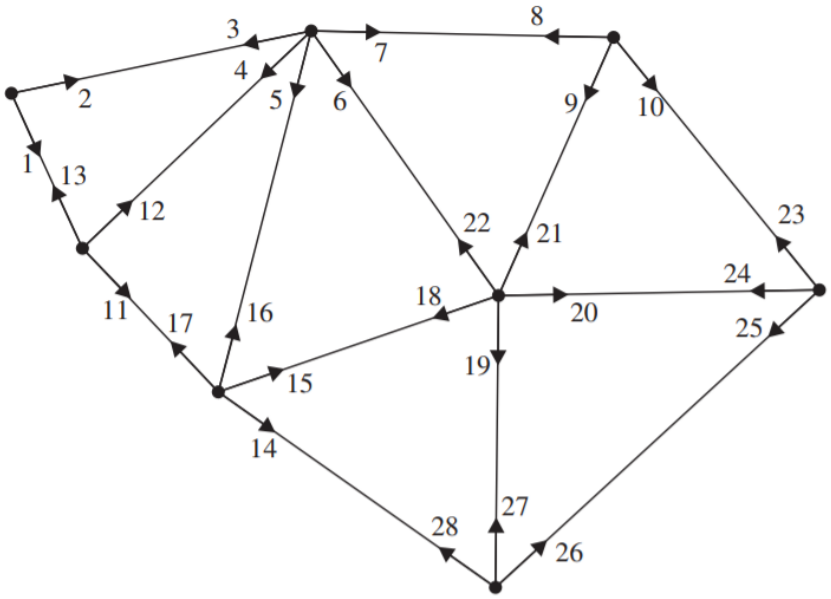


Figure 3.7: A visualization of a combinatorial map with the arrows representing the darts, from [DKWG08]

As this concave hull algorithm does not support such shapes, the point set is checked in advance. A simple algorithm calculates the slopes between consecutive points and compares them to another. If the differences are smaller than a margin the point set is considered a straight line. Then the two extreme points either in X or Y direction are used to form the path representing this line of seats.

3.4.2 ConcaveCube's concave hull algorithm

The second approach for the polygon computation is based on the concave hull construction method from ConcaveCubes[LCB+18]. The algorithm works on a clustered point set where each point is assigned a cluster. First a Delaunay Triangulation for the entire point set is generated. Then all edges which connect points of different clusters are deleted, followed by deleting all inner edges. Inner edges are the ones that appear twice in the triangulation result. Figure 3.8 visualizes the Delaunay Triangulation. The original algorithm is designed to only generate Jordan boundaries which are paths where each node has a degree of 2. After an edge deletion it can happen that some edges are exposed which leads to a non-Jordan boundary. If this is the case the algorithm removes and adds new edges so the property is ensured. In the case of seating plans non-Jordan boundaries are desirable. For example a line of seats should be represented as a single path instead of a polygon which can be examined in Figure 3.10. The slightly curved paths in the right image that appear on both sides represent a line of seats. The left image visualizes them as polygons, whereas the right image uses open paths. For the

seating plan image a path representation for a line of seats is preferred.

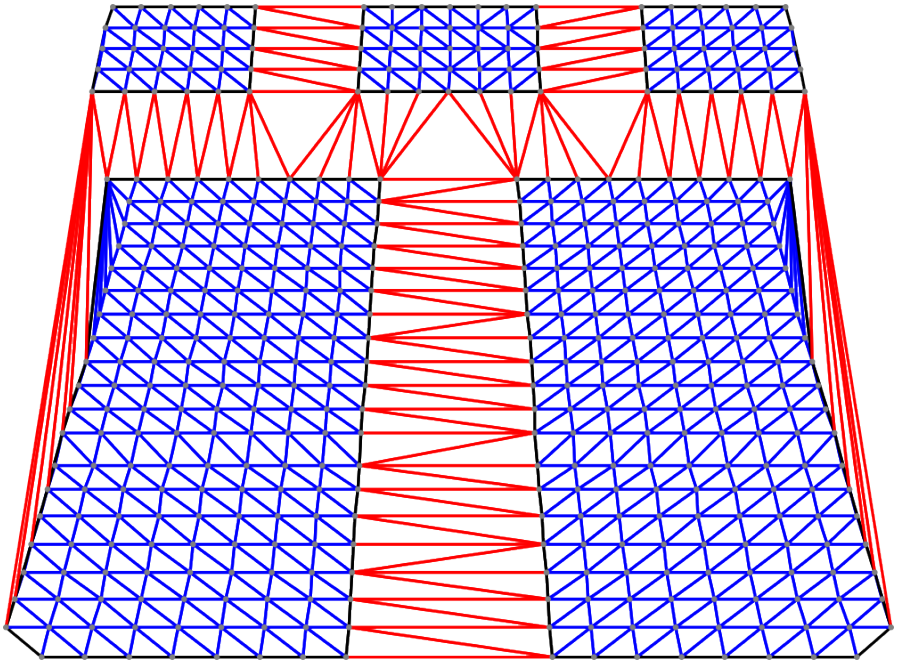


Figure 3.8: The Delaunay Triangulation of ConcaveCube's concave hull algorithm. Edges connecting different clusters are shown in red, inner edges are blue and boundary edges are black.

3.4.3 Comparison between the concave hull algorithms

Figures 3.9, 3.10, 3.11 and 3.12 compares the two concave hull methods side by side. Results of Duckham's algorithm are presented on the left side and results of ConcaveCube's algorithm are shown on the right side of each Figure. Both methods operate on the same clustering results. The first thing that comes to notice is the difference in polygon smoothness the algorithms produce. Duckhams's algorithm uses a smoothness input parameter to control the polygons' shapes. ConcaveCube's method does not take an input parameter, but the shape of the resulting polygons are based on the Delaunay triangulation. This leads to more edges and therefore can produce more complex shapes as seen in the middle polygon in Figure 3.10. For this project it is more desirable to be in control of the smoothness than not having to specify an input parameter.

The extension of ConcaveCube's concave hull algorithm that allows non-Jordan boundaries is preferred over the handling of seat rows in the other approach. When a curve of seats is represented by a polygon in Duckham's concave hull algorithm the intrinsic shape gets lost. This can be observed in the seat on both sides of the seating plan in Figure 3.10

Duckham's algorithm processes each point set individually and therefore overlaps of polygons can appear. This cannot happen with ConcaveCube's method because the

polygons are constructed simultaneoulys using the Delaunay Triangulation that does not have edge intersections and therefore no overlaps can appear.

For this project Duckham's approach is more preferred because of the ability to control the smoothness.

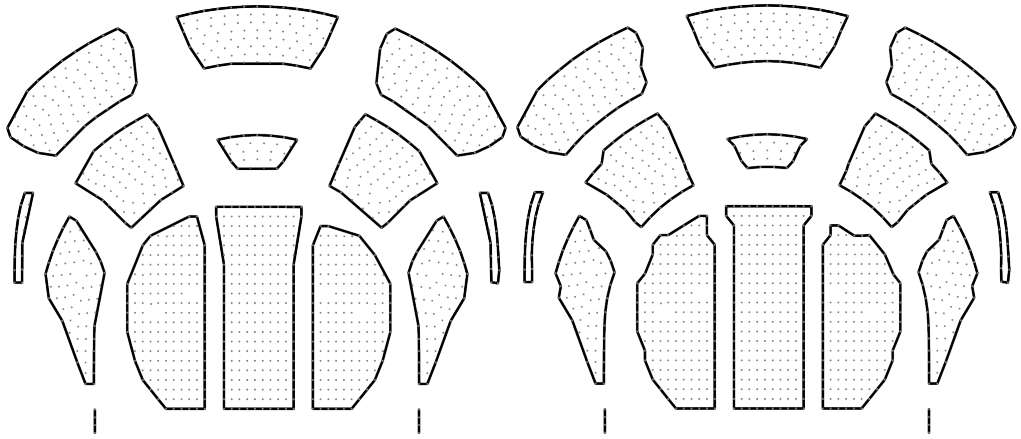


Figure 3.9: Comparison between Duckham's (left) and ConcaveCube's (right) concave hull algorithm

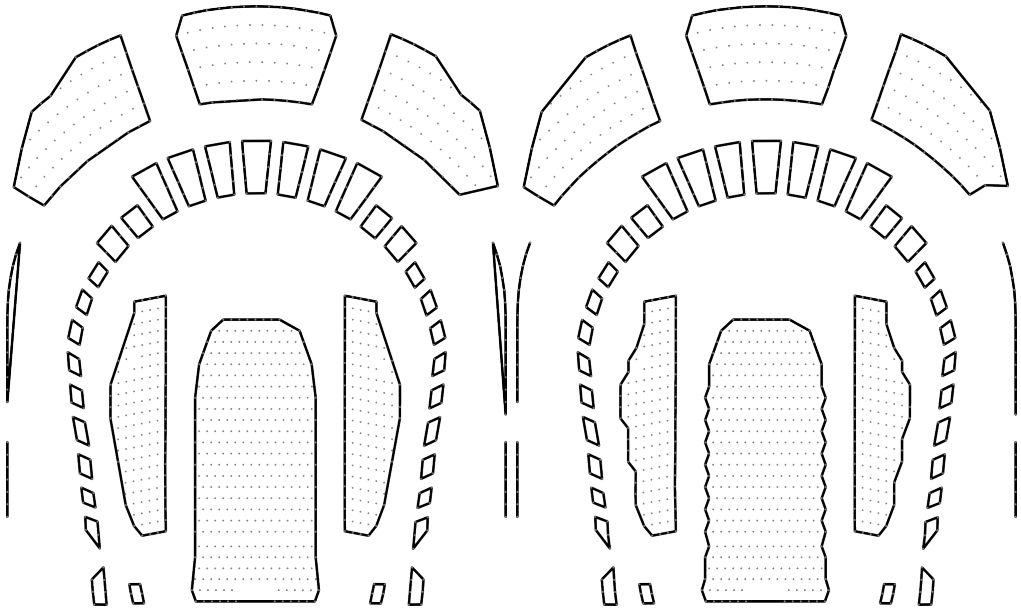


Figure 3.10: Comparison between Duckham's (left) and ConcaveCube's (right) concave hull algorithm

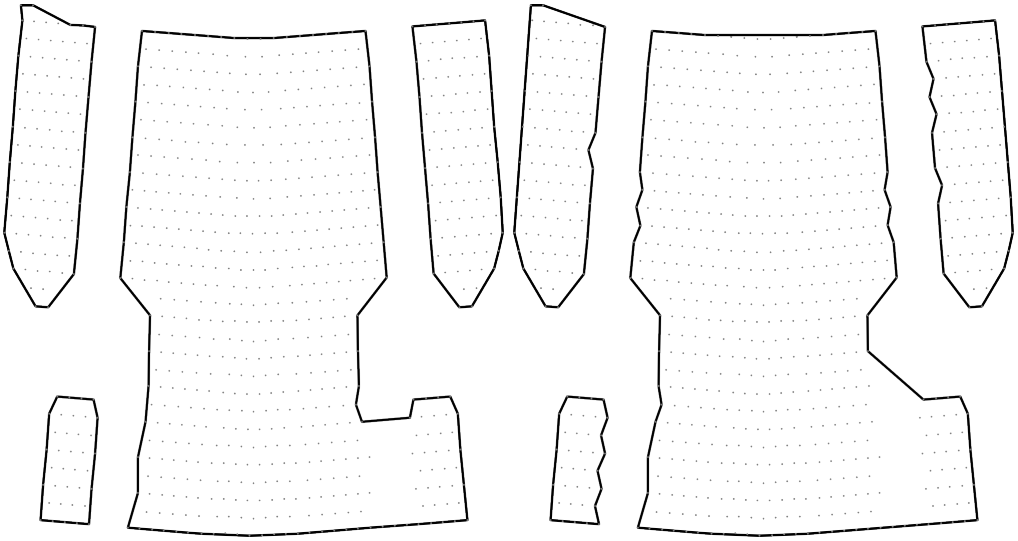


Figure 3.11: Comparison between Duckham's (left) and ConcaveCube's (right) concave hull algorithm

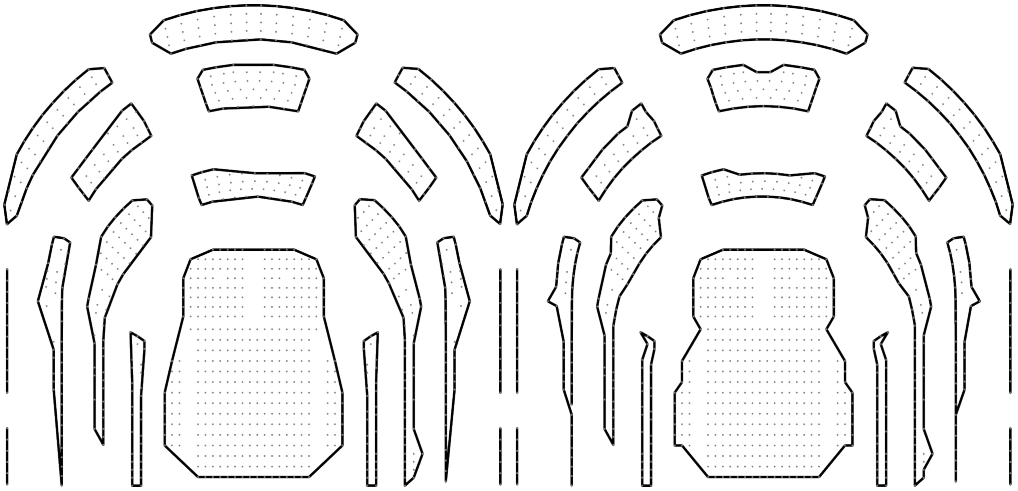


Figure 3.12: Comparison between Duckham's (left) and ConcaveCube's (right) concave hull algorithm

3.4.4 Calculating the stage's position

The position of the stage can not be derived from the data itself. By experience it is either below or above the seats. The user decides at the beginning to position the stage at the top or bottom. It will be centered horizontally. The calculation of the position of the stage can be seen from equations (3.3)–(3.6). A visual explanation for the variables can be found in Figure 3.13.

$$\text{width}_{\text{image}} = x_{\max} + x_{\min} + 2x_{\text{offset}} \tag{3.3}$$

$$x_{\text{stage}} = \frac{\text{width}_{\text{image}} - \text{width}_{\text{stage}}}{2} \tag{3.4}$$

$$y_{\text{stage(top)}} = y_{\min} + y_{\text{offset}} - \text{gap} \tag{3.5}$$

$$y_{\text{stage(bottom)}} = y_{\max} + y_{\text{offset}} + \text{gap} \tag{3.6}$$

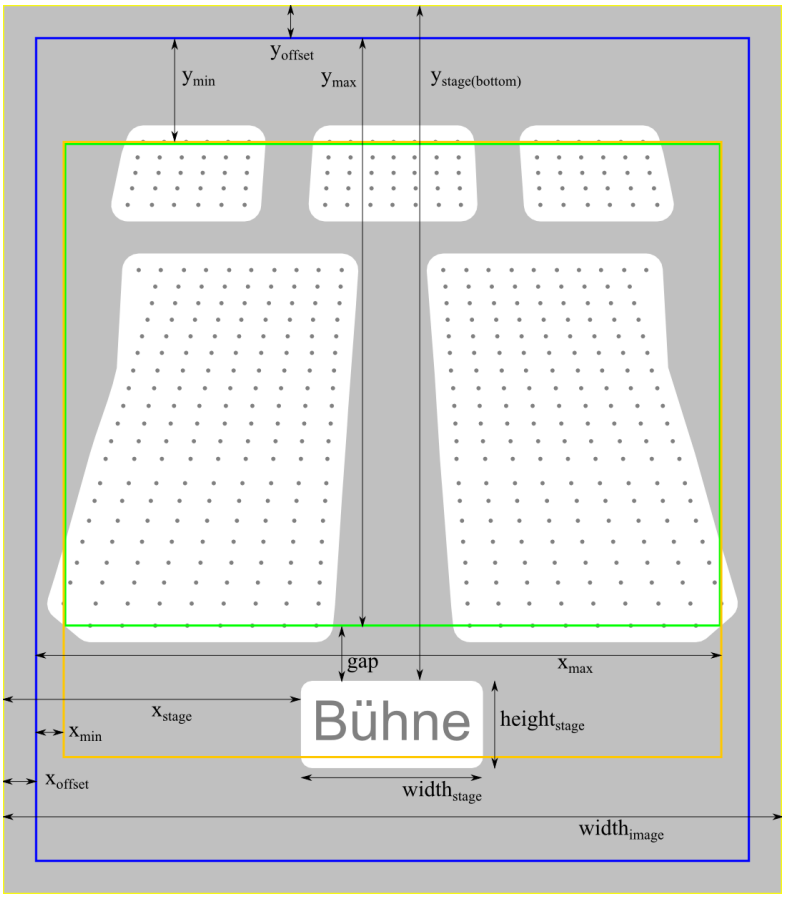


Figure 3.13: The created seat group polygons and the stage form the final image. The blue square represents the offset. The green square shows the minimum and maximum positions of seats and the orange square is the minimum and maximum including the stage.

CHAPTER 4

Implementation

The seating plan image generation is implemented on an existing system containing a client and server. The client is a website built with JavaScript and HTML and the server runs a Java Application. They communicate via a REST API. The user input, interactions and displaying of results is done on the client. The core algorithm for the seating plan image generation is implemented as a component on the server. Figure 4.1 gives an overview of the components and their relations.

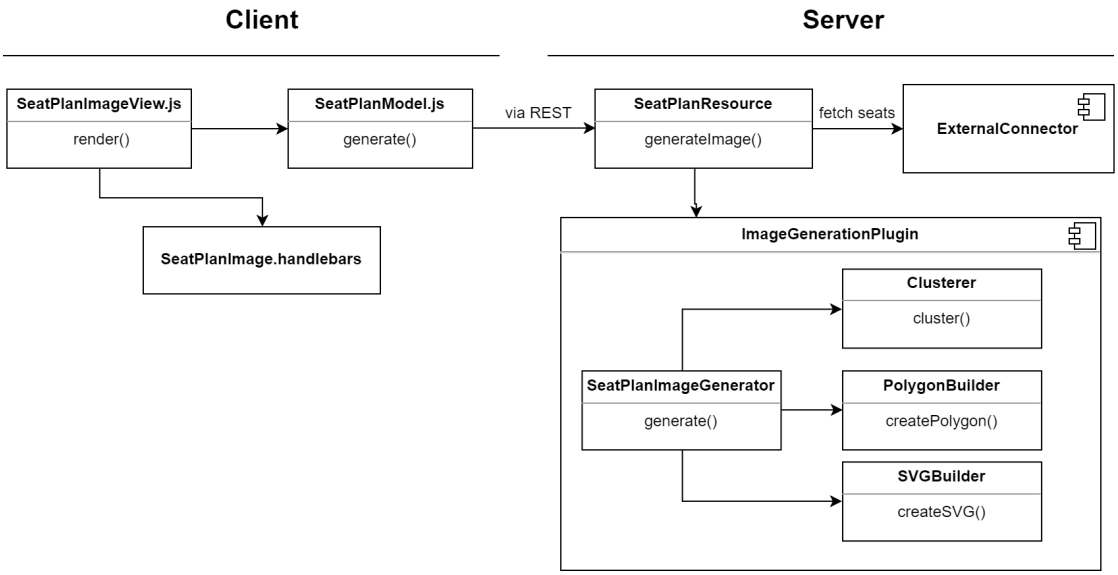


Figure 4.1: The component diagram showing the interaction between the client and server

4.1 User interface

In Figure 4.2 the user interface for the performance search and parameter input can be seen. The user can search an event using its name and other filter options. The left table displays all the events found. Clicking on an event will bring up all its performances on the right table. The user selects a performance for which the seating plan image should be generated from by clicking on it. Below the tables the input parameters for the algorithm are defined. They include the distance threshold *Eps* for clustering, the smoothness l for the polygon creation, x- and y-offset, and the options for clustering using the sector and box information. The input is predefined with values that were found to work for many seating plans. After all the input has been set, clicking the generate button will initiate the image generation.

In Figure 4.3 the generated seating plan image is displayed on the left. The stage can be repositioned using drag-and-drop. Dragging the polygon will move the whole stage including the font. The font can also be dragged alone. On the right side the input for manipulating the stage can be found. This includes sliders for the width, height, font size and stroke width. Additionally the dots representing the seats can be hidden, since they will be removed for the final image used in the app. Also the display of guides can be toggled which show the offset, center, maximum and minimum horizontally and vertically. At the bottom of Figure 4.3 the buttons are located for downloading, saving, and uploading the image.

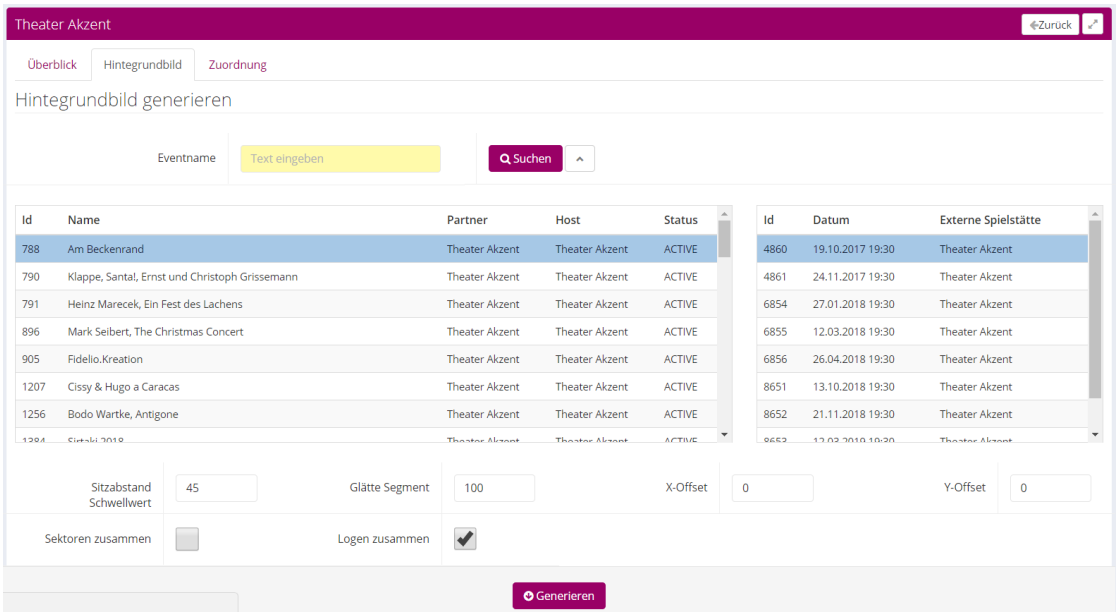


Figure 4.2: The user interface to search for a performance and input parameters that are used to generate the seating plan image.

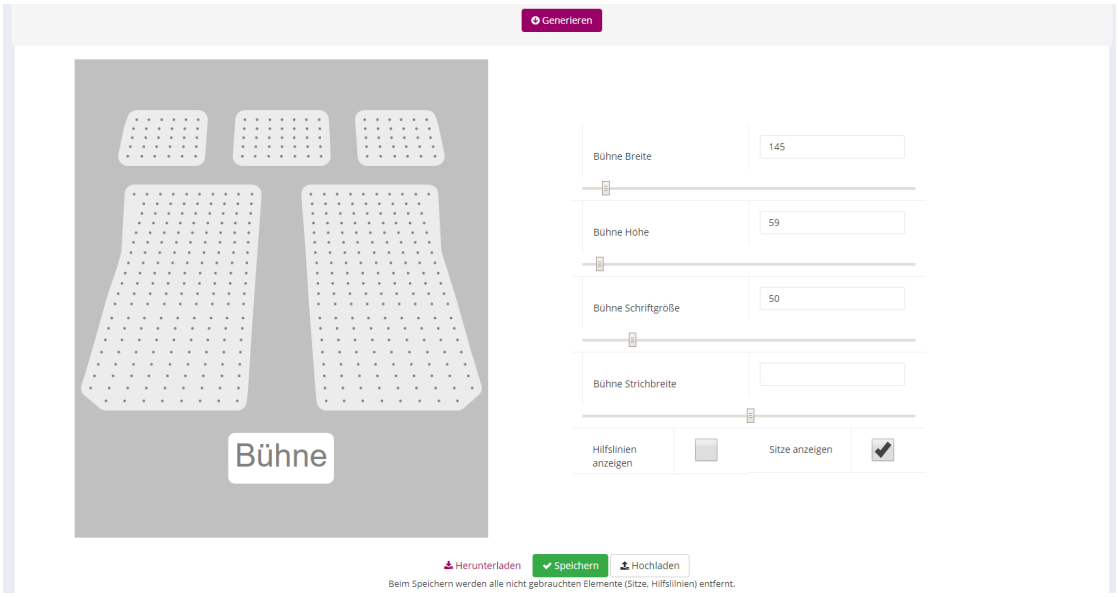


Figure 4.3: The user interface showing the generated seating plan image. On the right side are the controls to manipulate the stage. Additionally the stage can be repositioned via drag-and-drop.

4.2 Seat data

The retrieval of the seat data for a performance is done on a dedicated component outside the scope of this project. This component fetches the data from external sources and converts it to Java Objects.

4.3 Clustering

The clustering is done with the Java library *Commons Math: The Apache Commons Mathematics Library* [Apa]. The library offers among others an implementation of the DBSCAN algorithm which is used for this project. The algorithm is used by providing a wrapper class for a seat which implements the Clusterable interface and using a custom distance measurement class.

4.4 Polygon computation

The first approach for computing the polygons which is based on Duckham's algorithm [DKWG08] was implemented using *Ophensphere Project's* [Gro12] concave hull implementation. The second approach based on ConcaveCubes's algorithm[LCB+18] was implemented using a Java Delaunay Triangulation library [Die18].

CHAPTER 5

Results and Discussion

It is found that the automated seating plan generation can produce sufficient results to replace the manual creation process. Many venues have a simple seat layout with evenly spread out seats which can easily be clustered using only the position. The box information for clustering is needed for seating plans like in Figure 5.1. The seats that form a horseshoe shape belong to several boxes. The evaluation of the polygon computation showed that smooth polygons are desired which can be achieved with the approach using Duckham's concave hull algorithm[DKWG08]. The next sections will look into how different parameter values will effect the result.

5.1 Tuning the distance threshold *Eps* for clustering

When clustering using the seat's position The distance threshold effectively controls the size of the seat groups in terms of the number of seats. It determines the maximum distance between two seats to be considered neighbours, hence falling into the same cluster. Smaller values result in more smaller seat groups whereas greater values lead to fewer but bigger seat groups. Figure 5.2 shows seating plan images generated with different values for the distance threshold *Eps* for Equation (3.1). An *Eps* value of 30 to 45 had been found suitable for all tested seating plans.

5.2 Tuning smoothness parameter of polygons

The smoothness controls the coarseness of the polygon shape in Duckham's concave hull algorithm[DKWG08]. Small values result in more complex shapes than greater values. The value determines the maximum length of the polygon's longest edge. Figure 5.3 shows seating plan images generated with various smoothness values. For most seating plan images a smoothness value of 100 had been found suitable, because the seating plans

are of similar dimension. For the testing data a minimum value of 30 is needed to get a reasonable image. Values greater than 100 did not lead to significant changes.

5.3 Comparison to manually created images

Comparing the existing seating plan images that had been created by hand to the generated images show how similar-looking results can be achieved. Figure 5.4 compares the seating plan of *Theater in der Josefstadt*.

The process of creating the seating plan images manually is very time consuming. For one seating plan image to be finished takes about 20 min. In comparison the average time of generating the image takes about 3 min including testing different parameters and modifying the stage. The person in charge that creates the seating plan images tested the final product and said: "It already saves me time - 5 minutes instead of an hour for the Schauspielhaus".

5.4 Limitations

Figures 5.5 to 5.10 show several generated seating plan images of different complexity. All results use Duckham's concave hull algorithm[DKWG08] for the polygon computation. When seats are not evenly spread out like in Figure 5.6(a) it is difficult to determine the distance threshold Eps. In this case having a larger value would merge the 4 lines in the lower left corner, but would also merge all other seat groups. Figure 5.8(a) indicates the problem of overlapping seat groups. The central 3 polygons are too nearby so their borders touch. The simple solution here would be to reduce the border thickness, but the algorithm does not detect this on its own. Another issue is that the smoothness setting affects all polygons the same. It is not possible to have some polygons with a more complex border than others. Also the implementation only allows modification of the stage. If anything in the image should be modified it must be done with an external tool.

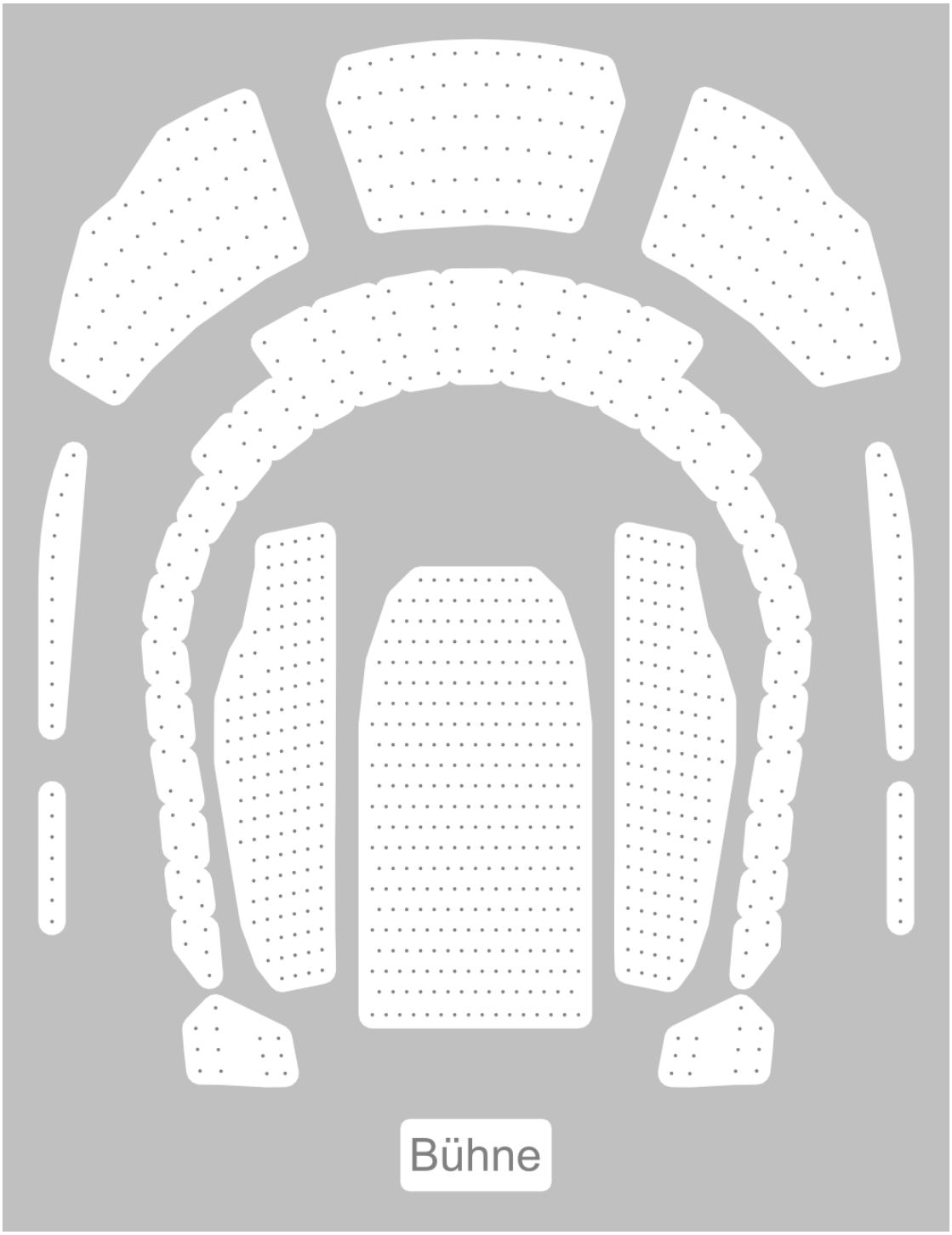
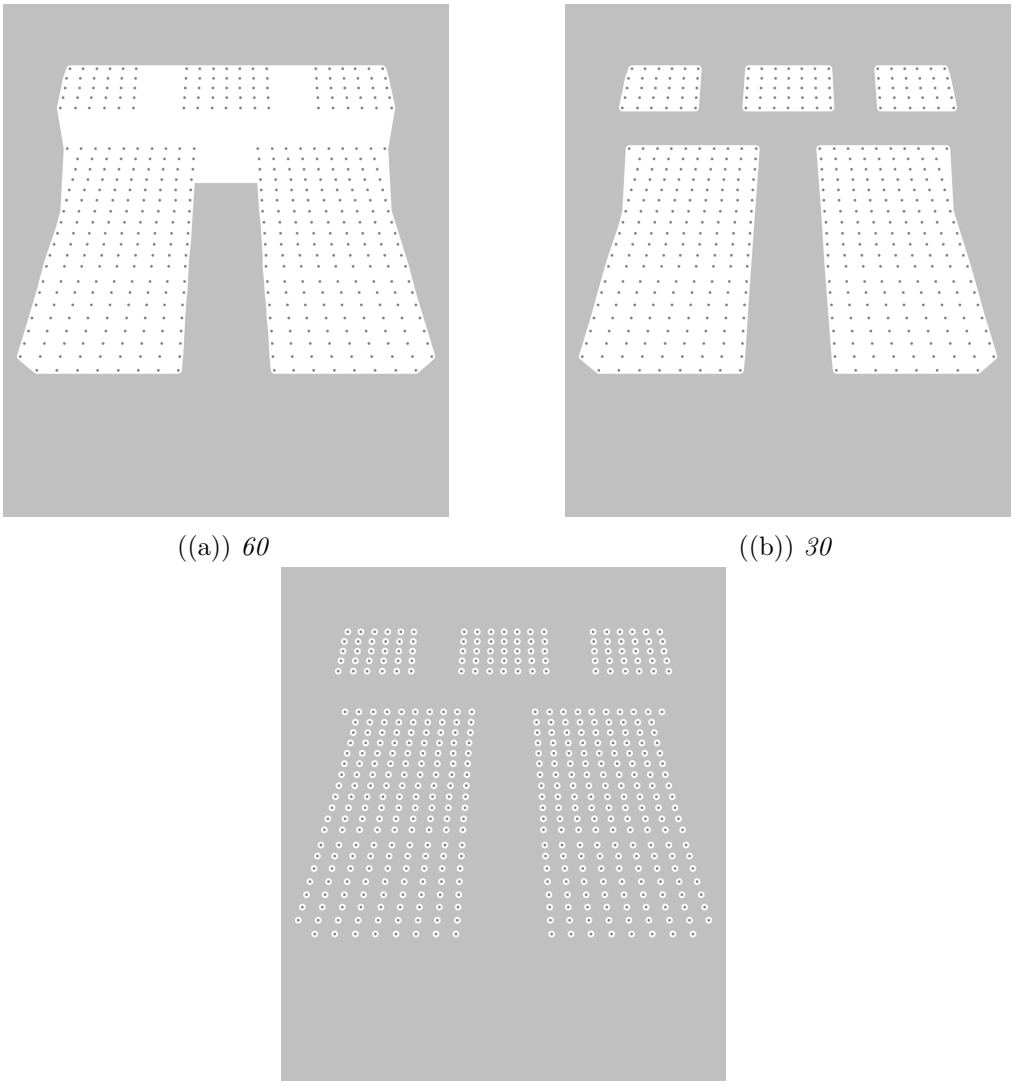


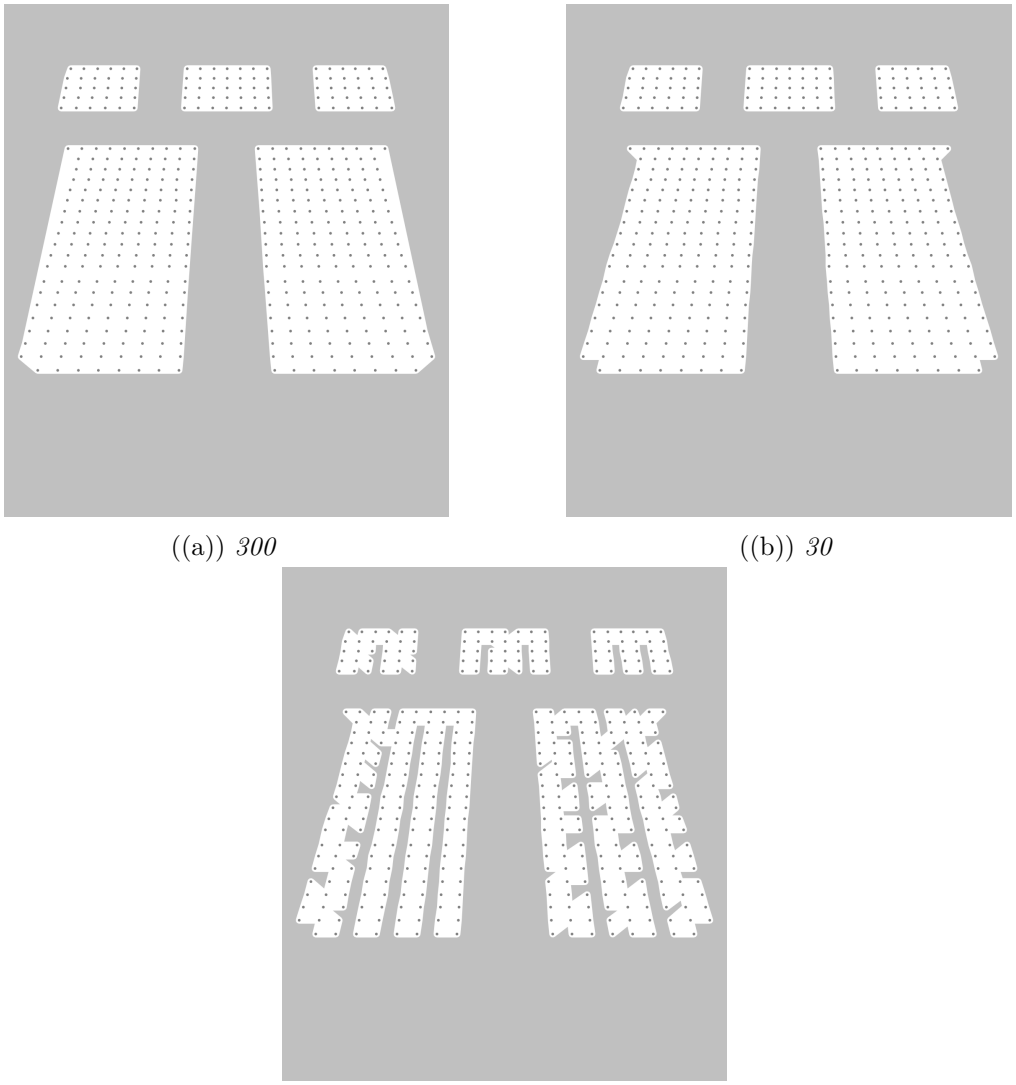
Figure 5.1: Generated seating plan image of the venue *Ronacher*.



((a)) *60* ((b)) *30*

((c)) *5*

Figure 5.2: Using different values for the *distance threshold*. It determines the maximum distance between two seats to call them neighbours and therefore being able to be clustered together. The smoothness is set to 100.

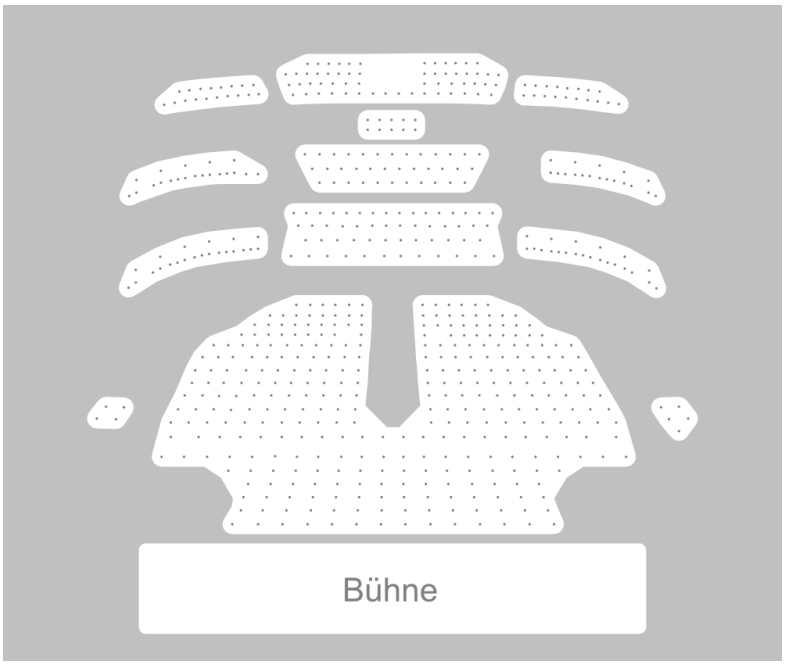


((a)) *300* ((b)) *30*

((c)) *5*

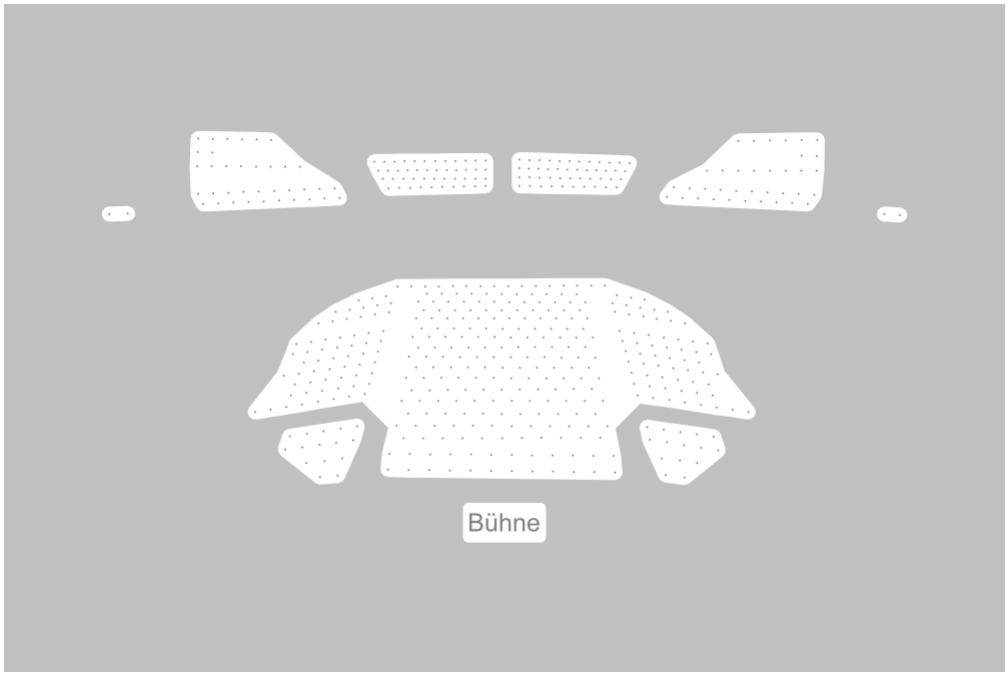
Figure 5.3: Using different values for the *polygon smoothness*. It determines the maximum length of the longest edge. The clustering is done via position only and a distance threshold of 45.

((a)) *Hand drawn*

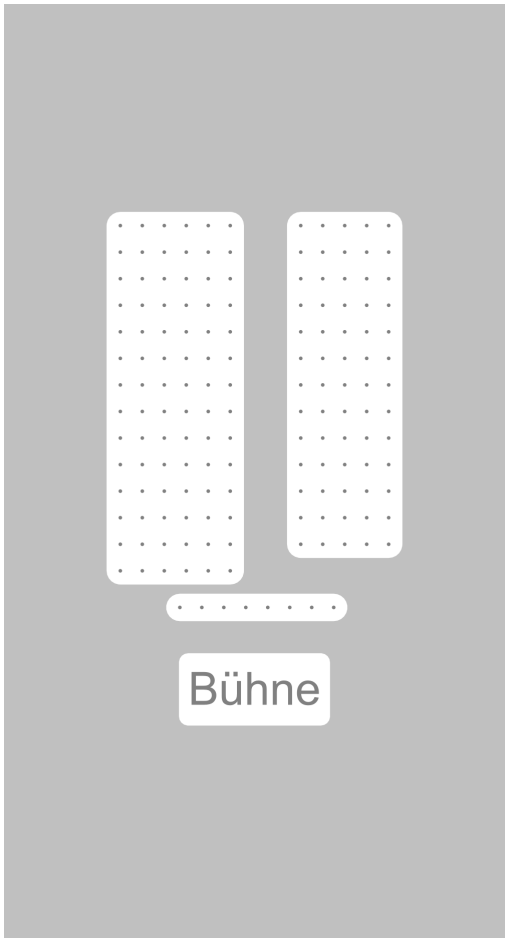


((b)) *Generated*

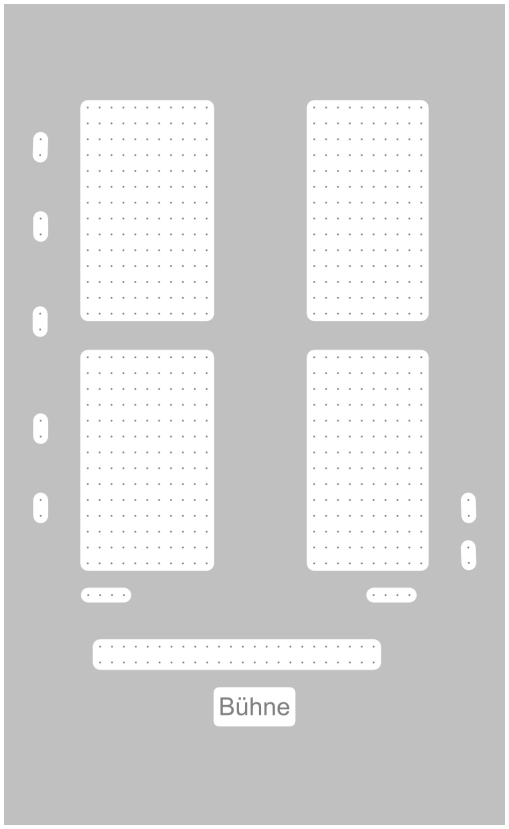
Figure 5.4: The comparison between the manually created and generated seating image depicts how similar results can be produced.



((a)) Akademietheater

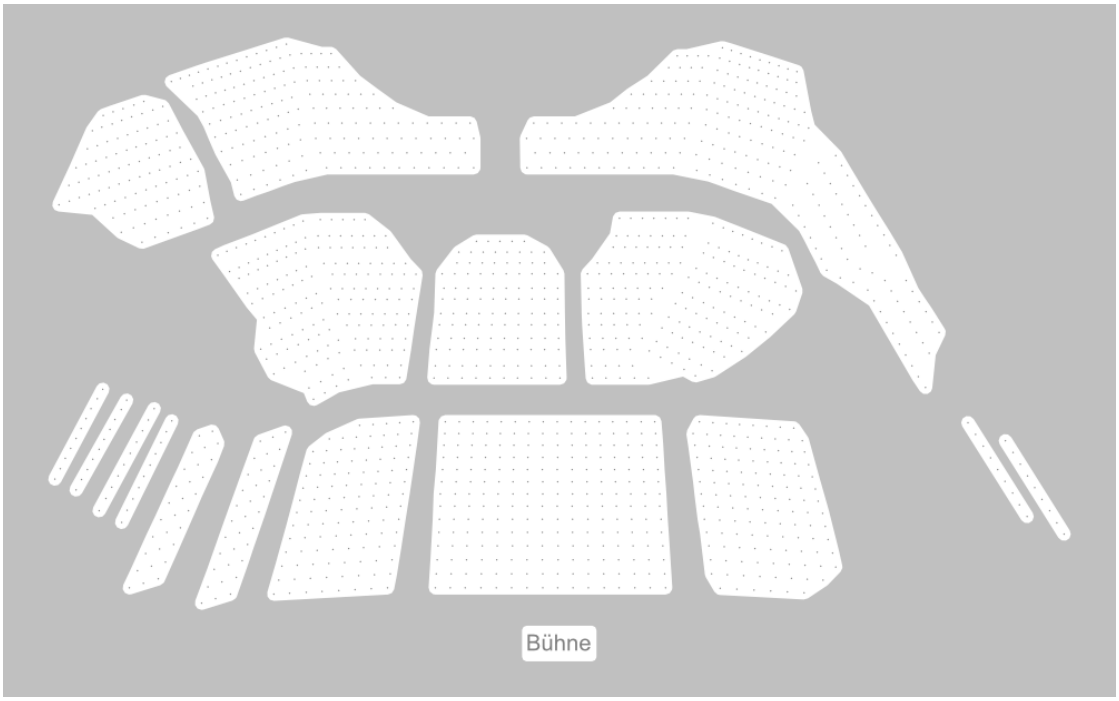


((b)) Schloss Esterhazy, Empiresaal

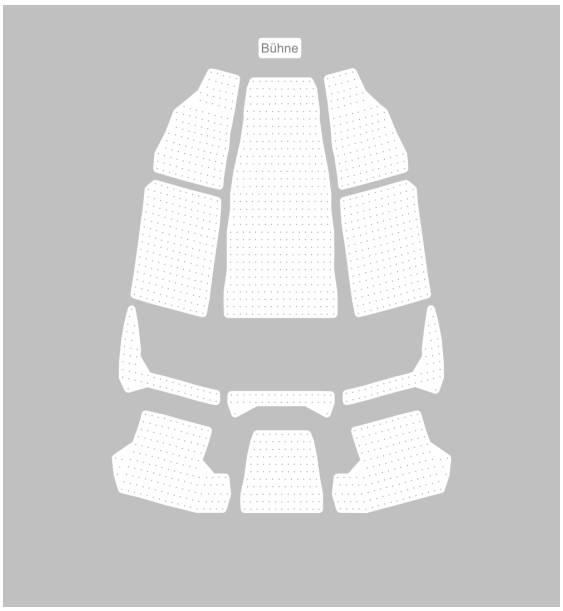


((c)) Schloss Esterhazy, Haydnsaal

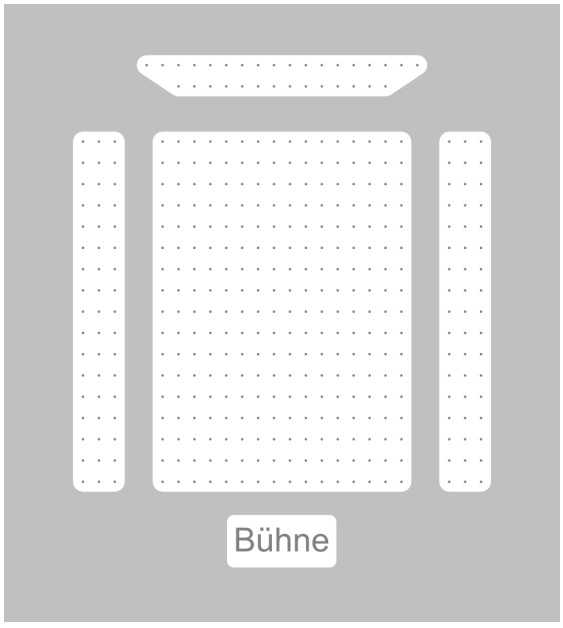
Figure 5.5: Generated seating plan images.



((a)) Congress Innsbruck

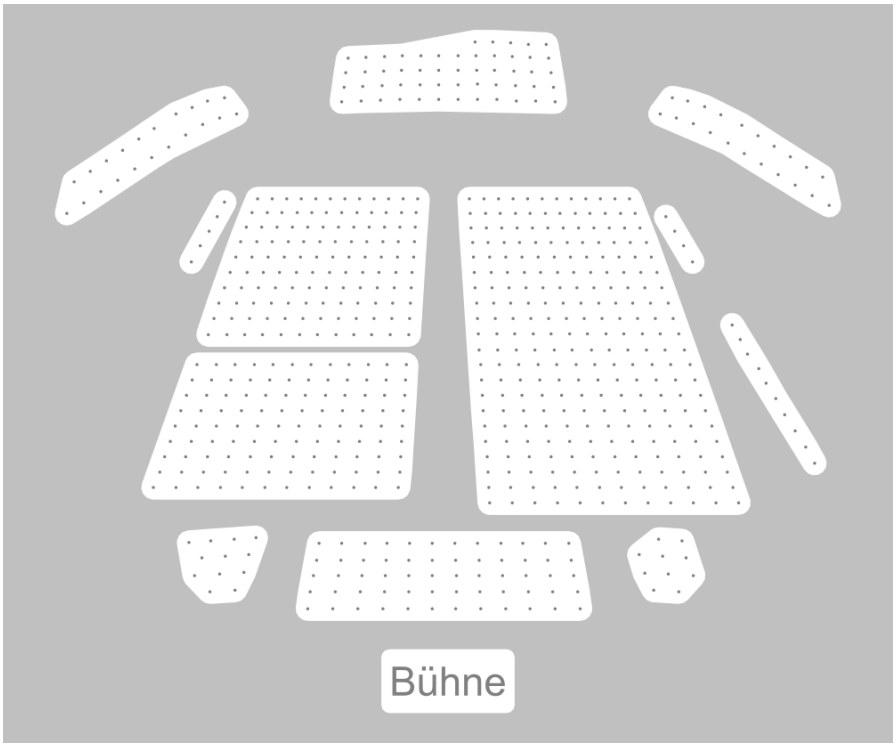


((b)) Festspielhaus Bregenz

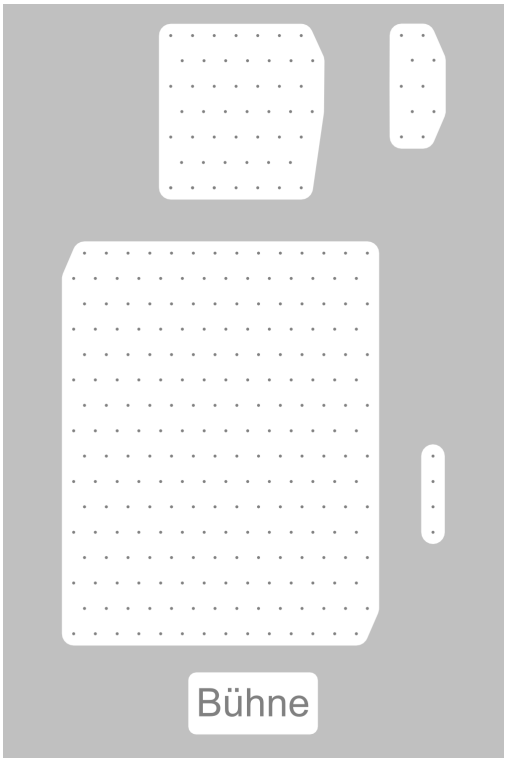


((c)) Haus der Musik Innsbruck

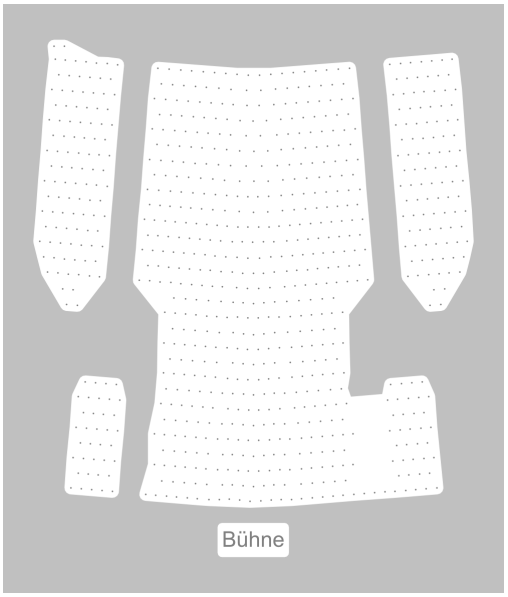
Figure 5.6: Generated seating plan images.



((a)) Renaissancetheater

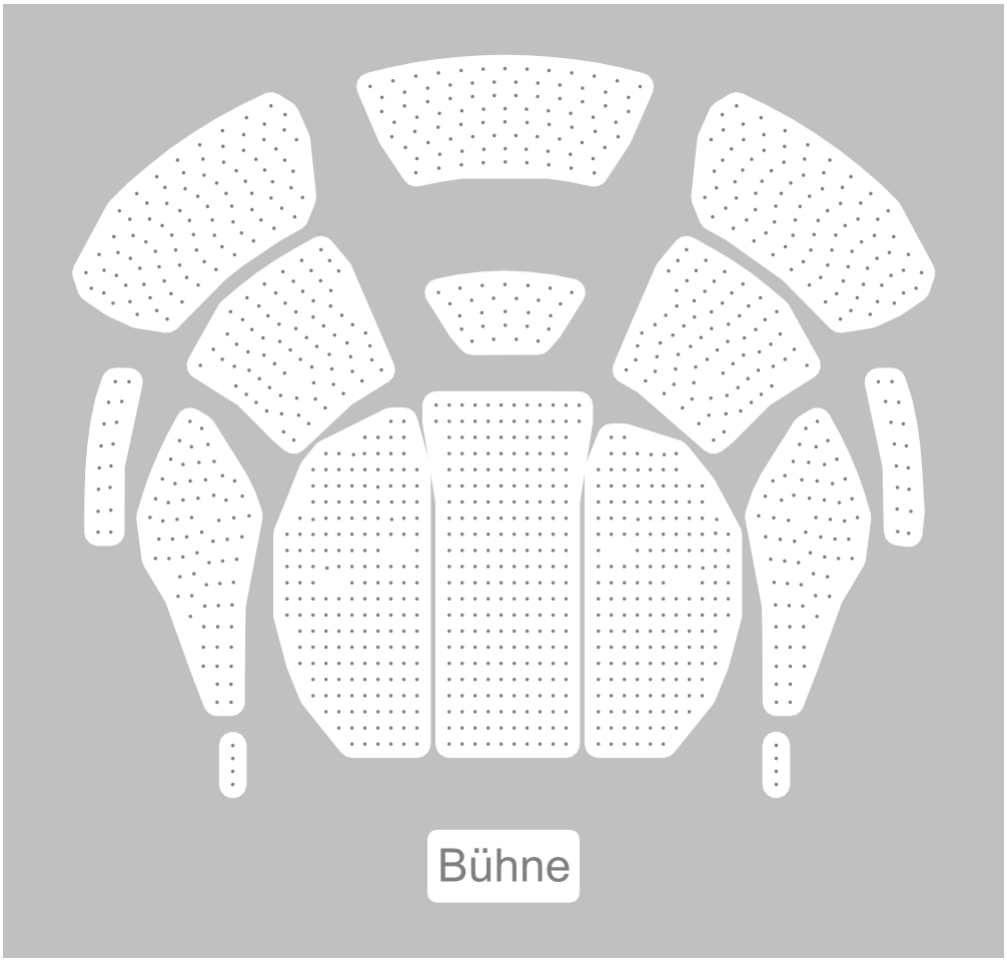


((b)) Kammeroper

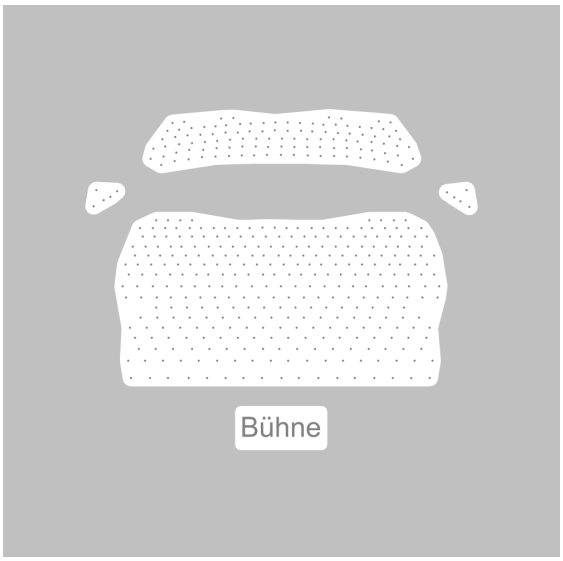


((c)) Schloss Tabor

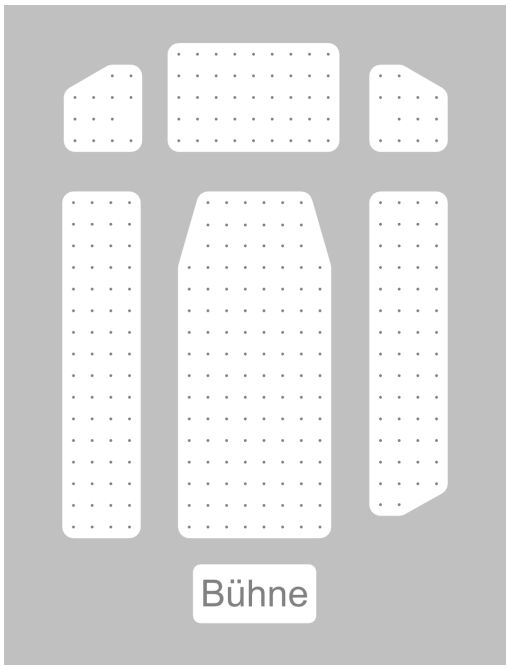
Figure 5.7: Generated seating plan images.



((a)) Raimund Theater

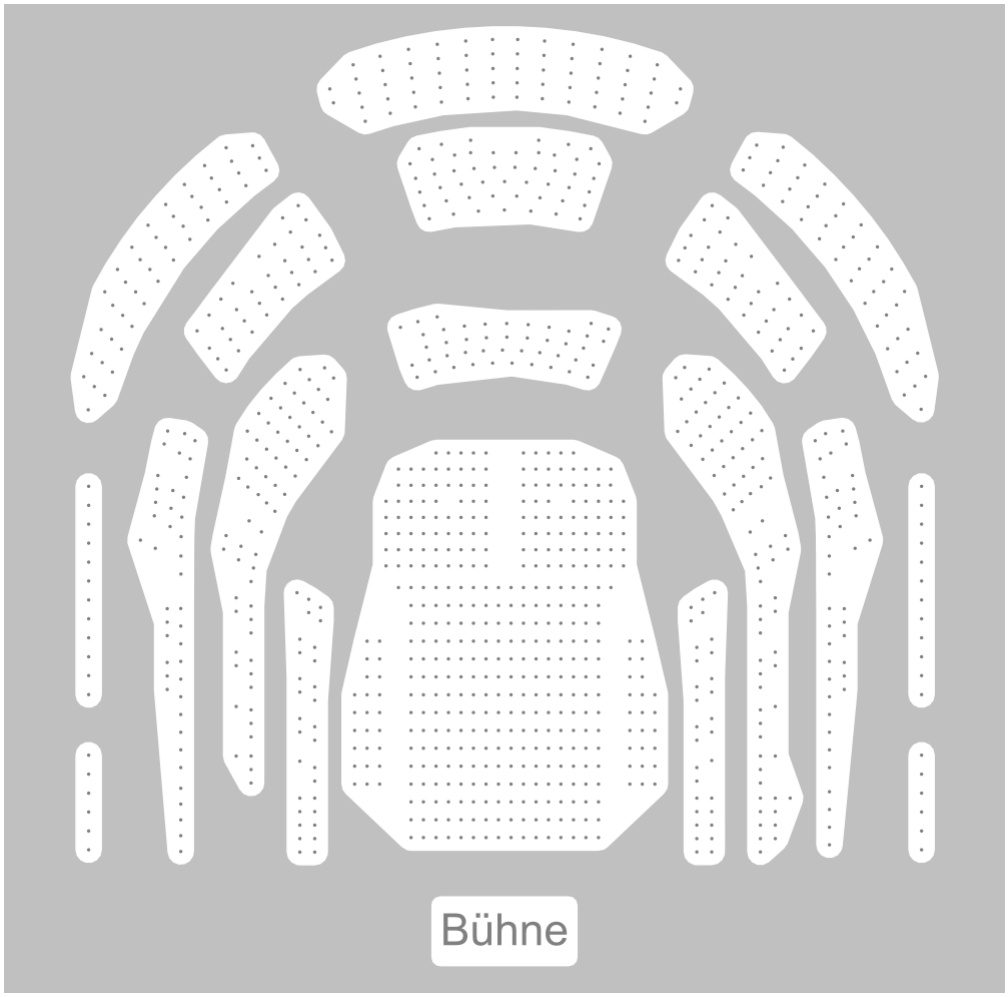


((b)) Kammerspiele der Josefstadt

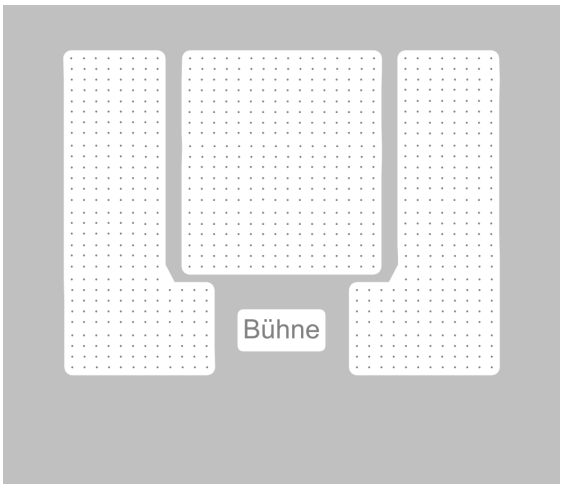


((c)) Tiroler Landeskonservatorium

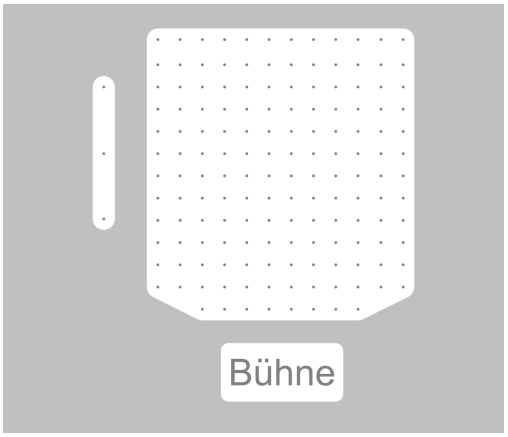
Figure 5.8: Generated seating plan images.



((a)) Theater an der Wien

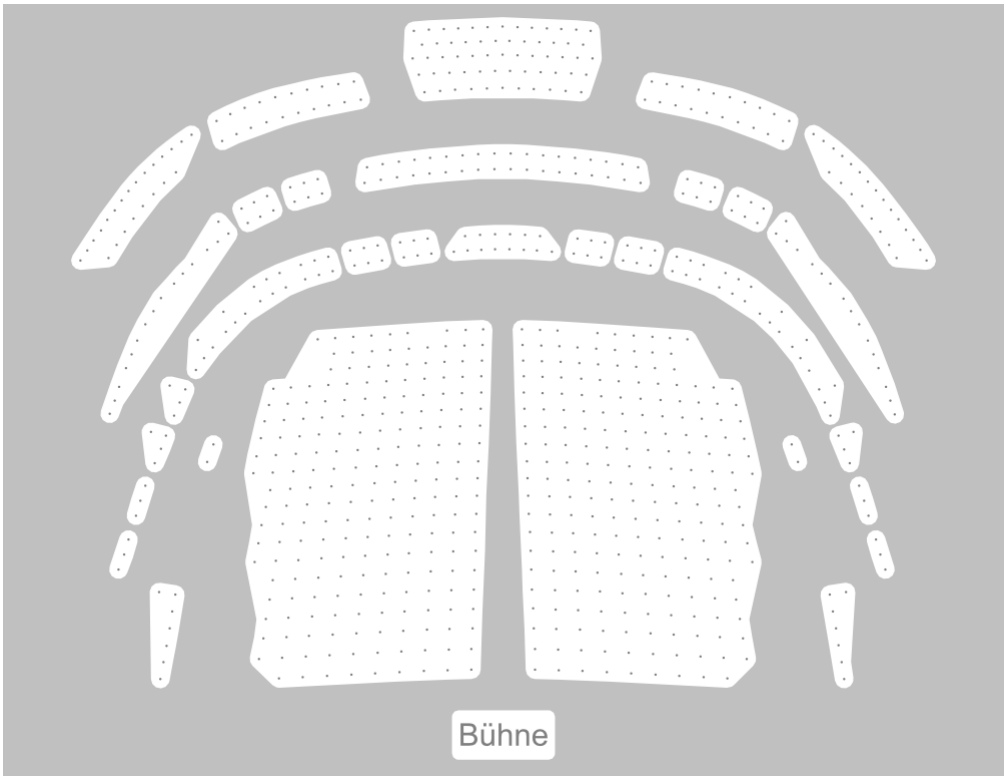


((b)) Schloss Kobersdorf

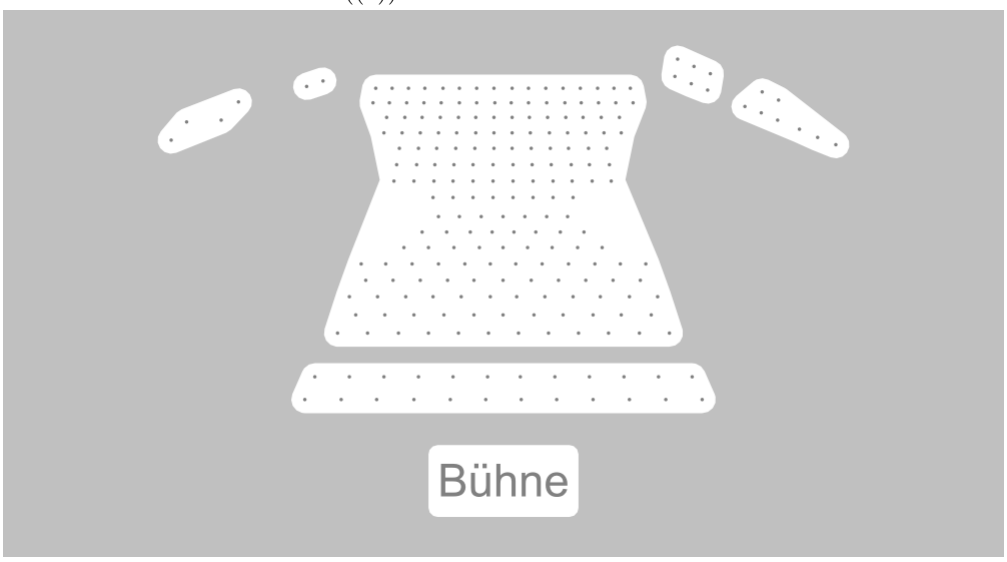


((c)) Sträußelsäle

Figure 5.9: Generated seating plan images.



((a)) Tiroler Landestheater



((b)) Theater im Zentrum

Figure 5.10: Generated seating plan images.

CHAPTER 6

Conclusion and Future Work

The developed product can automatically generate seating plan images out of seat information like in Figure 5.1. The goal of replacing the manual creation of these images with an automated process that manages to produce similar results is achieved. The seat groups are generated through DBSCAN clustering mainly with the position information. It was found that an average threshold can be used for most seating plans. Two approaches for computing the polygons that represent seat groups have been compared. ConcaveCube's concave hull algorithm[LCB+18] does not need an input parameter and can produce non-overlapping polygons. Duckham's concave hull algorithm[DKWG08] uses an input parameter that allows controlling the smoothness of the polygons which has been preferred over the other approach. The finished product can handle most of the seating plans and is sufficient enough to be used in the real work environment. The time spent on creating seating plan images could be reduced significantly. Future work would be to eliminate the limitations mentioned in section 5.4.

During the development the question arose if it would be possible to detect if a created seating plan still matches a seating layout. Sometimes it is the case that a seat layout change for an event is undetected and the seating plan image does not fit anymore. For example new seats have been added which are not covered by the polygons anymore. An algorithm could detect such cases and automatically generate a new seating plan image for the event.

Another improvement would be if the user could control the clustering by placing walls which would separate seat groups. For this the modified seating plan image needs to be used as an input for the new image generation.

List of Figures

Bibliography

[Apa] Commons Math: The apache commons mathematics library. http://commons.apache.org/proper/commons-math/. Acessed: 2018-09-07.

[Die18] Johannes Diemke. A 2d delaunay triangulation library for java. https://github.com/jdiemke/delaunay-triangulator, 2018. Acessed: 2018-09-07.

[DKWG08] Matt Duckham, Lars Kulik, Mike Worboys, and Antony Galton. Efficient generation of simple polygons for characterizing the shape of a set of points in the plane. *Pattern recognition*, 41(10):3224–3236, 2008.

[EKS+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[Gro12] Eric Grosso. Opensphere concave hull. https://github.com/atolcd/pentaho-gis-plugins/blob/master/concave-hull/src/main/java/org/opensphere/geometry/algorithm/ConcaveHull.java, 2012. Acessed: 2018-09-07.

[Jar73] Ray A Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information processing letters*, 2:18–21, 1973.

[Kim14] Stanley H Kim. Systems and methods for generating dynamic seating charts, July 10 2014. US Patent App. 14/148,354.

[LCB+18] Mingzhao Li, Farhana Choudhury, Zhifeng Bao, Hanan Samet, and Timos Sellis. Concavecubes: Supporting cluster-based geographical visualization in large data scale. In *Computer Graphics Forum*, volume 37, pages 217–228. Wiley Online Library, 2018.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[MC12] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[MS07] Adriano Moreira and Maribel Yasmina Santos. Concave hull: A k-nearest neighbours approach for the computation of the region occupied by a set of points. 2007.

[seaa] SeatAdvisor the smart ticketing solution. `https://www.seatadvisor.com/online-ticketing/`. Accessed: 2018-05-08.

[seab] seatsio.io. `https://www.seats.io/demos/designer`. Accessed: 2018-05-08.

[soc] socialtables. `https://www.socialtables.com/`. Accessed: 2018-05-08.

[tic] Ticket gretchen. `https://ticketgretchen.com/`. Accessed: 2018-05-08.

[XW09] Rui Xu and Donald C Wunsch. Clustering. hoboken, 2009.