Stable 3D City Layout in ExplorViz

Bachelor’s Thesis

Alexander Barbie

September 30, 2014

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Reinhard von Hanxleden
Prof. Dr. Wilhelm Hasselbring
M.Sc. Florian Fittkau
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,
Abstract

Software landscapes in companies keep growing and become more and more complex. Today several hundreds of single applications can build such a software landscape. Since most of the software landscapes grew over several years, often the knowledge about the communication and utilization of a landscape is lost due to a lack of documentation. Without a well documented communication between the applications, it is almost impossible to understand all the connections and interactions inside a software landscape.

ExplorViz is a tool for live trace visualization. The visualization of entities and traces inside an application is layouted by the rectangle-packing algorithm, which only takes nodes but no edges into account. In this thesis, we will show that the rectangle-packing algorithm is not stable, since its object arrangement depends on the size of an entity. We will propose a new layouting algorithms for ExplorViz that is based on quadtrees for the entity placement and a graph for the edge layouting. The graph builds a grid structure by the aid of the borders of quadtrees. Shortest paths between entities will be computed by Dijkstra’s shortest path algorithm. To enhance the quadtree’s space efficiency, empty nodes will be cut off. This cutted quadtree approach is more stable than the rectangle-packing algorithm and can be used as an alternative layouting algorithm in ExplorViz, under some restrictions.
Contents

1 Introduction .................................... 1
   1.1 Motivation .................................... 1
   1.2 Document Structure ......................... 2

2 Goals ............................................. 3
   G1: Implement an Application-level Layout in ExplorViz .......... 3
   G2: Evaluation Concerning the Performance of the Layout Algorithm .......... 3
   G3: Evaluation Concerning the Stability of the Layout Algorithm .......... 3

3 Foundations and Technologies .................... 5
   3.1 ExplorViz ..................................... 5
   3.2 Graphs ....................................... 5
   3.3 Stability ..................................... 8

4 Our Layouting Approach .......................... 11
   4.1 Metaphors .................................... 11
   4.2 Quadtrees .................................... 12
   4.3 Requirements for the Algorithm ............... 13
   4.4 Steps ........................................ 14
      S1: Entity Placement ........................... 14
      S2: Layouting of Edges ....................... 15

5 Entity Placement ................................ 19
   5.1 Placement Algorithm ......................... 19
      5.1.1 Calculation of Package Sizes ............ 20
      5.1.2 Insertion ................................ 21
      5.1.3 Cutting ................................ 24
   5.2 Stability .................................... 26
      5.2.1 Preparation .............................. 27
      5.2.2 Stability Implementation ............... 27

6 Layouting of Edges ................................ 29
   6.1 Edge Layouting Algorithm .................... 30
      6.1.1 Pin Creation ............................. 30
      6.1.2 Paths Building ......................... 31
      6.1.3 Finding Shortest Paths .................. 32
Software landscapes in companies keep growing and become more and more complex. Since most of the software landscapes grew over several years, often the knowledge about the communication and utilization of a landscape get lost due to a lack of documentation. Today several hundreds of single applications can build such a software landscape. Without a well documented communication between the applications, it is almost impossible to understand all the connections and interactions inside a software landscape. With size and complexity of such systems, the amount of code increases. Therefore, it is not advisable to gain knowledge about the software landscape by analyzing the code of each application. [Fittkau et al. 2013]

Due to the enhancement of computer performance and broadband internet, new methods allow recovering the knowledge about the communication and utilization of a software landscape. Such a new method is live trace visualization used, for instance, in ExplorViz. It observes method calls and saves monitoring records that are combined to traces. Based on these traces, ExplorViz visualizes the collected data and communication from single applications in software landscapes as a 3D city model in a browser interface.

1.1 Motivation

[Fittkau et al. 2013] present a tool for live trace visualization called ExplorViz. The visualization of the traces inside an application is layouted by the rectangle-packing algorithm by [Wettel 2010], which only takes nodes but no edges into account. A communication between two classes is represented by a line between them. It crosses other visualization entities and becomes confusing the more traces are present (see Figure 1.1). Further, the rectangle-packing algorithm is not stable, because the object arrangement depends on the size of an entity. If an entity increases or decreases in size, it will be placed completely different and all following entities, too. A inconsistent placement of objects makes it confusing to follow communications and counteracts the purpose of easy reverse engineering. This bachelor thesis deals with the lack of clarity by finding an algorithm for a stable 3D city layout. Only a stable algorithm affords an observer the greatest possible benefits from our 3D city layout. A randomized arrangement of nodes would become confusing, even if traces do not cross other visualization entities.
1. Introduction

![Diagram](image.png)

**Figure 1.1. Crossing method calls**

1.2 Document Structure

This bachelor thesis consists of 9 chapters. Chapter 2 describes our goals. Afterwards, Chapter 3 presents important foundations and used technologies of the thesis. Thereafter, Chapter 4 sketches the approach to fulfill the goals detailed in Chapter 2. The implementation of the placement algorithm will be given in Chapter 5 and Chapter 6 deals with the layouting of edges. A performance and stability evaluation of our implemented algorithm is presented in Chapter 7. In Chapter 8 related work is illustrated and in Chapter 9 a conclusion and future work will be given.
The purpose of the thesis is to find an algorithm for a stable 3D city layout in ExplorViz. To reach this goal, sub-ordinate targets will be aimed for. In the following sections these goals will be presented.

**G1: Implement an Application-level Layout in ExplorViz**

Finding a stable algorithm for a 3D city layout is one goal of the thesis. The layout shall look like a 3D map, with roads that do not run through buildings. Therefore, the algorithm has to avoid edges that cross entities by using an alternative route around obstacles instead of connecting entities directly. Further it has to be stable between two layoutings, i.e., no object and trace is arranged by random. For example, an object that was placed in the upper left corner shall be placed there again in future layoutings. Therefore, an algorithm has to stick to proper arrangement guidelines which will be defined in more detail in the thesis.

**G2: Evaluation Concerning the Performance of the Layout Algorithm**

New algorithms for entity placement and the layouting of edges should, in best case, be more performant than the predecessors. Under no circumstances, they should be much worse in their performance and efficiency. Therefore, both versions, the new and the current used algorithm, will be evaluated concerning their performance. The following criteria will evaluated in Goal 2:

- runtime
- compactness
- squareness of the map
- edge length
2. Goals

**G3: Evaluation Concerning the Stability of the Layout Algorithm**

After implementing a stable 3D city layout algorithm, it should be evaluated concerning its quality. It is not sufficient to find a stable algorithm for only one application, it has to be evaluated with several other applications, e.g., EPrints, to ensure the stability between several layoutings for as many applications as possible. The following criteria will evaluated in Goal 3:

- stability between layoutings
Chapter 3

Foundations and Technologies

The focus of this thesis is the implementation and evaluation of a new, stable, 3D city layouting algorithm in ExplorViz. We assume the reader has knowledge about city metaphor, software cities [Knight and Munro 2000] and treemaps [Shneiderman 1992]. [Steinbrückner 2012] and [Wettel 2010] describe the two latter approaches. [Tauer 2009] evaluates the EvoStreet approach concerning its stability and gives a good overview of all relevant terms.

This chapter describes the foundations and technologies that will be used in the following chapters. To not to go beyond the scope of this thesis, the given definitions and explanations are compressed to the most important, in this thesis used, facts. For more detailed information, we refer to the given literature.

3.1 ExplorViz

ExplorViz is a tool to visualize monitoring data from software landscapes in a browser interface. It is developed by Florian Fittkau [Fittkau et al. 2013] and intends to support reverse engineering without the need to examine source code of each application. An user gets an overview of the whole software landscape in a 2D view or a detailed 3D view of a single application in the landscape.

The 2D view shows software clusters that consist of different applications. These applications are connected by lines to visualize which applications communicate with others (Figure 3.1). By double clicking on an application, the user can browser through the applications and sees the 3D view of one applications (Figure 3.2). The 3D view uses the cite metaphor to visualize communications and the control flow between classes. We can highlight packages or classes to show only communications relevant to them. When mouse hovering a communication a small pop up shows which functions are called between the connected entities.

3.2 Graphs

A graph is an often used data-structure in computer science. Thus, we assume that graphs and their basic operations are known by the reader and we just give some short definitions to show which type of graph is used in this thesis.
3. Foundations and Technologies

**Figure 3.1.** 2D view of a software landscape

**Figure 3.2.** 3D view of a single application

**Definition 3.1 (Graph):**

>A graph (or undirected graph) is a pair $G = (V, E)$, with a finite set $V \neq \emptyset$ and a finite set $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$.

>A directed graph (or digraph) is an ordered pair $G = (V, E)$ with a finite set $V$ and a finite set $E \subseteq \{(u, v) \mid u, v \in V, u \neq v\}$.
3.2. Graphs

(a) Directed Graph  
(b) Undirected Graph

Figure 3.3. Drawing of the graph $G_1$ in (a) and of an undirected graph $G_2$ in (b)

Elements $v \in V$ are called vertices and pairs $e = \{u, v\} \in E$, $u, v \in V$ are called edges.

Example 3.1.1:
Be $G_1 = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 4), (4, 3), (2, 1), (5, 2)\}$. Be $G_2$ like $G_1$ but with undirected edges. The drawing of both graphs is shown in Figure 3.3.

Definition 3.2 (Dijkstra’s Shortest Path Algorithm):
In [Dijkstra 1959] an easy to implement and today common shortest path algorithm was described the first time. This definition is based on this description but is adapted to a more efficient solution in the thesis later. The algorithm does the following:

1. To find the shortest path between two vertices, the weight of a path is calculated as the sum of the weights of the edges between the source and target vertex.
2. A path is a shortest path, iff there is no path from the source vertex to a target vertex with lower weight.
3. The shortest path from a source vertex to a target vertex will be found by increasing the distance from the source vertex.
4. The algorithm chooses the edge with the lowest weight, stores this value and adds the next minimum value from the next edge it selects.
5. We start at one vertex and branch out by selecting certain edges that lead to neighbor vertices.

Definition 3.3 (Orthogonal Embedding of Graphs):
A graph layouting is orthogonal, if and only if it exists only of vertical and horizontal lines.
3. Foundations and Technologies

Requirements for Good Graph Drawing

A graph drawing should in no way confuse the observer, instead it should highlight the special characteristics of a given graph. The crucial factors for this intention are the algorithms that compute, create, and finally draw the graph. Although, everyone has a different opinion on what is confusing and what is not, [Ambler 2005] gives some criteria for good graph drawings:

- Avoid crossing lines
- Depict crossing lines as a jump
- Avoid diagonal or curved lines, place symbols on a grid
- Node distance
- As short as possible edge length

3.3 Stability

ExplorViz’s purpose is to support users in reverse engineering by visualizing an application’s structure and interactions, instead of examine code. When a user browses through the 3D model of an application, he will learn about its structure and he will try to understand the connections and calls inside this application. [Misue et al. 1995] named this effort, to become familiar with the structure, "build a mental map". Since ExplorViz is a live trace visualization tool, applications can change while runtime, i.e. new classes appear, and the user has repeatedly to adjust his mental map. Therefore, it would be advantageous to support an user in structure recognition and minimize the effort in building a mental map [Kaufmann and Wagner 2001]. In other words, for a good understanding the layout should not change, even if classes appear or disappear. This structure recognition is the main aspect for stability of algorithms.

To evaluate the stability of layoutings a, more precisely, definition of stability is needed first. Accordingly, we lean on a definition given in [Mannl 2010]. Based on the fact that a layouting can be considered as a graph, with entities as vertices and traces as edges, two consecutive layoutings can be compared, with regard to their similarity. One before and one after changes occurred. Each layouting method gets a value of similarity this way. The layouting algorithm with an higher grad of similarity is more stable than the one with a lower grad.

Closely connected with the definition of stability, is the metric that is used to take measurement of the stability of layoutings. To compare layoutings, equal criteria for comparison have to be found. Since all layouting algorithms we use have in common, that they place geometrical shapes, especially rectangles, on a two-dimensional fundamament, we use this analogousness to take measurement of similarity [Mannl 2010].
3.3. Stability

Similarity

As mentioned, all layouting methods have in common that they place geometrical shapes on a two-dimensional fundament. Vertices of graphs are represented by points in space and will be presented as rectangles in layoutings. In the rectangle-packing algorithm and also in our approach, rectangles are computed during layout creation. They influence the positions of different entities, respectively rectangles, directly. Therefore, only one point of a vertex will be taken for comparison. In this thesis it will be the center point of rectangles, to avoid different results due to unequal measuring points.

Since the task is to place geometrical objects on a two-dimensional fundament, we operate in the Euclidean space and can use the Euclidean metric, which allows the usage of all familiar arithmetics for two- and three-dimensional coordinates. Although, the definition of Euclidean space is familiar, we commemorate to the Euclidean norm and angular measure.

**Definition 3.4 (Euclidean Norm and Distance):**

*Be\( x, y \in \mathbb{R}^n, n \in \mathbb{N} \).* The Euclidean norm of \( x \) is

\[
||x|| = \sqrt{x_1^2 + \cdots + x_n^2}
\]

The distance between \( x \) and \( y \) is

\[
||x - y|| = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}
\]

**Definition 3.5 (Angular measure):**

*Be\( x, y \in \mathbb{R}^n, n \in \mathbb{N} \).* The angular \( \angle(xy) \) is defined as

\[
\angle(x, y) = \arccos \frac{x \cdot y}{||x|| \cdot ||y||}
\]
Chapter 4

Our Layouting Approach

Although, there are different approaches for visualization of software systems via software cities [Wettel 2010] [Steinbrückner 2012], none of them can be adopted by ExplorViz. ExplorViz visualizes software cities on a very abstract way and needs a consistent layouting algorithm. The current layouting algorithm, the rectangle-packing-algorithm by [Wettel 2010], is not consistent, since the placement order depends on the size of entities. If a new component will be placed or a large component deleted, the whole layout changes as a result of the relocation of all following entities. Furthermore, an efficient edge layouting that fulfills all requirements is not possible, too. The algorithm just determines coordinates for entities but does not save the unused space. We simply cannot avoid collisions on an elegant way, since we cannot detect them.

[Steinbrückner 2012] presents a more stable approach, the EvoStreets approach. Its fishbone like arrangement of software evolutions is more stable than the EvoStrees approach by [Wettel 2010], especially since new streets are added, if new entities evolve. Unfortunately, its disadvantage is its lack of lucidity for users. An evolution of viewed software cities is easy to follow, but communications between entities are just sparsely to follow.

In the following sections the approach will be described, detailed implementations will be given in Chapter 5 and Chapter 6.

4.1 Metaphors

Due to a lack of alternative layouting algorithms that can be adopted by ExplorViz, a new approach is needed. Our approach borrows from two metaphors. [Knight and Munro 2000] motivate in their Software World the usage of a city metaphor for the visualization of the static architecture of software systems. ExplorViz leans on the city metaphor of [Wettel 2010]. The city metaphor defines the metrics how artifacts are visualized, but not how they are arranged. In Section 4.4 we will introduce the hierarchical setup of ExplorViz. The representation of this hierarchy is a crucial factor for the expressiveness of a layouting. Establishing a mental map with little effort is a challenge, when using the city metaphor. Thus, layoutings should be reduced to the essential information and only semantic loaded content visualized. [Smallman et al. 2001] figures out users prefer a 2D representation, as they provide higher information availability than 3D representation. This representations often borrow from the map metaphor, which allows the user to thinking of this metaphor
4. Our Layouting Approach

and establish a mental map [Schulz et al. 2011].

We mentioned the hierarchical setup of ExplorViz already. This hierarchy is reflected in the tree structure how visualization entities are managed in order to be layouted, namely component trees. Traditionally trees are represented by, rooted, directed graphs. Their root node is at the top of the page and children nodes below the parent node with lines connecting them. Although this represents hierarchy on a basic level and allows establishing a mental map with minimum effort, this kind of representation does not reduce the layouting of software systems to their essentials. ExplorViz needs to display package and class structures. Packages are nodes with classes as leaves and hence they would be represented as roots of subtrees. This displays hierarchy in a clearly way. However, the essential information of packages in programming languages is only a grouping of classes to their tasks. Thus, root nodes would claim more attention and space in the layouting than they provide on relevant information. An alternative representation to display hierarchy in a space efficient way, was given by [Shneiderman 1992] in his treemap approach.

Treemaps represent hierarchical structured data, like trees are, as nested areas. Root nodes are, usually, represented as rectangles and child node rectangles completely cover their parental rectangle representation. As a consequence, this kind of representation can guide the attention to the contained classes, but still shows the package structure of a software system. Accordingly, we will use a data structure that can represent treemaps to arrange the visualization of entities in ExplorViz. The differentiation in hierarchy will be provided by the usage of the space in all three dimensions and the city metaphor. Packages are the foundation on which classes are put and displayed apart from each other.

4.2 Quadtrees

A quadtree is a tree data structure in which each node has exactly four children [Finkel and Bentley 1974], namely \{Northwest, Northeast, Southeast, Southwest\}. Its purpose is to partition a two-dimensional space by recursively subdividing it into four quadrants. Therefor, the root of a quadtree represents the drawing board and each node is a partition that is a new quadtree itself. In favour of a more handy usage in our algorithms, we define a special quadtree called region quadtree, which is presented in [Samet 1984]. Region quadtrees divide their roots in four, in size equal, child nodes. Thus, we can put all entities in individual nodes and save all used and also unused spaces, of the entire drawing board. Quadtrees are usually used for managing spatial data, like maps. Furthermore, they are used for chip architecture (Very Large Scale Intergration [Mead and Conway 1980]), since they can easy partition space. Accordingly, quadtrees are suitable for our approach to arrange entities by treemaps.

Definition 4.1 (Quadtree):
A tree $Q = (V, E)$ is called quadtree, if and only if:

\[
\Rightarrow \text{Each node has a maximum of four, in size equal, children.}
\]
4.3 Requirements for the Algorithm

- Children of nodes are marked with $m \in M = \{ \text{Northwest}, \text{Northeast}, \text{Southeast}, \text{Southwest} \}$.
- All children of the same parent node are marked different with $m \in M = \{ \text{Northwest}, \text{Northeast}, \text{Southeast}, \text{Southwest} \}$.
- Exclusive nodes without children can contain entities, at the utmost one.

An example is shown in Figure 4.1.

![Quadtree data structure](image1)

**Figure 4.1.** Quadtree data structure

### 4.3 Requirements for the Algorithm

The main task of this thesis is finding a stable 3D city layout algorithm. To meet the minimum requirements it has to fulfill:

1. **Stability**
   The algorithm has to be stable regarding its object/edges arrangement.

2. **Squareness is still given**
   Squareness of the layouting is a requirement

3. **Compactness**
   Compactness means a layout uses its space as efficient as possible and free space is minimized. Compact structures can increase the recognition value of layoutings.

4. **As short as possible edges**
   A simple approach to avoid traces that cross obstacles could be routing the traces around the whole map. This would be more confusing than traces crossing other objects. Expressiveness and effectiveness of a layout are important to be able to display all characteristics inside an application. To fulfill this intention traces have to be displayed as short as possible edges.
4. Our Layouting Approach

5. Avoid edges crossing obstacles
The algorithm has to avoid edges that cross obstacles by using an alternative route around obstacles instead of connecting entities directly. In no way edges are allowed to cross entities, however, they still allowed cross each other. These crossings have to be minimized to ensure software engineers benefit from the most possible lucidity of our layout.

4.4 Steps
The introduction to ExplorViz in Section 3.1 shows that users can switch between a 2D landscape-view of an entire software-landscape and a 3D view (leaned on the cite metaphor) of a single application in that software-landscape. This thesis exclusively deals with layouting algorithms using the city metaphor.

All the data we use is saved in an application object. An application contains its components and clazzes, which are saved in a tree structure, and all communications between the clazzes. Components are like nodes, they can contain further components and also clazzes. On the contrary, clazzes are like leafs of the tree. Both types of entities have in common that they can be drawn the same way. Hence, they both inherit from the class Draw3DNodeEntity. Communications are always between clazzes. However, if a clazz’s parent component is closed, only the communication between a clazz and the target’s parent component is shown. This hierarchal setup of ExplorViz we have to keep, when designing a new layouting algorithm. As overview, the package structure is given in the UML diagram in Figure 4.2.

By reason of the hierarchal setup, the entire layouting of ExplorViz will consist of different parts that are closely interrelated. Amongst other tasks these parts calculate the size of each entity, their placement sequence, their coordinates on the drawing board, and finally the layouting of edges.

S1: Entity Placement
One goal of this bachelor’s thesis is a new entity placement algorithm in ExplorViz. Why ExplorViz needs a new algorithm for entity placement, was already addressed before. In this section we only propose the parts of our approach. Our new algorithm will not be as space efficient as the rectangle-packing algorithm. Nevertheless, we will try to make it more stable from scratch. Therefor, a quadtree like in Definition 4.1 will be implemented and these six activities followed:

1. Sizes of all entities have to be calculated. First clazzes get a fixed size, afterwards the sizes of all components will be calculate. The corresponding quadtrees will get the size of components, since components are foundations on which clazzes are placed on. As a result, each component creates a new drawing board, respectively quadtree, to place entities on.
2. To determine their coordinates on the drawing board, all entities will be inserted into the quadtree.

3. After insertion, all components back up their latest bounds to their old bounds to save their size, since the next steps could influence that sizes. These old bounds are needed for future layoutings.

4. When all entities on one level are inserted, all empty nodes of the quadtree will be cut off. Conducting this at each hierarchy level, the free space will be reduced.

5. After cutting of the quadtree as far as possible, each component saves its corresponding quadtree and new bounds. Furthermore, the component saves its insertion sequence of all entities it contains, also for future layoutings. This is important to create fixed structures for the purpose of an high recognition value of a layouting.

6. All future layoutings compare the latest calculated sizes of all components with their corresponding old bounds and decide which size has to be used for the new layouting. Additionally, all entities will be placed in the order they were saved in the insertion order list. New evolved entities will be placed at the list’s end.

**S2: Layouting of Edges**

One more, quite complex, aspect of this thesis is a new edge layouting. ExplorViz’s current edge layouting exists of a line between center points of two entities (see Figure 1.1). In the
4. Our Layouting Approach

context of our placement algorithm, a new edge layouting procedure will be implemented. Build on quadtrees, we try to use their structure to find shortest paths between entities, without edge-entitity or edge-edge collisions. Again, this section just shows which steps our algorithm takes. A more detailed implementation will be given in Chapter 6.

As mentioned, this approach builds on the structure of quadtrees. More particularly, it uses them as a grid on which all edges proceed. Figure 4.1b insinuates how a grid could look like. This approach is based on graphs. Each quadtree creates at least eight anchor points, which will be called pins in this thesis. These pins will be the graph’s vertices and connected pins the graph’s edges. Pins only can be connected with their neighbors. Otherwise, the embedding is not orthogonal and edges that do not cross entities are not ensured. An orthogonal embedding allows to leave out collision detection procedures and helps to improve the clarity of a layouting [Purchase et al. 2001].

Twice contained edges in the graph can cause errors during path findings and slow down the layouting. Thus, it is important to avoid equivalent edges in the graph. After all pins and edges were created, each entity allocates out-going pins. A shortest path will be computed between this out-going pins, by Dijkstra’s Shortest Path algorithm (DSP) (Definition 3.2). Since ExplorViz draws lines between two points, the pins DSP identifies will the added to a list to let them connect by ExplorViz’s current implementation. Summarized:

\( \Rightarrow \) First a grid structure will be created, with the aid of quadtrees.

\( \Rightarrow \) All pins will be added as vertices, as well as all connections between as edges, to the graph.

\( \Rightarrow \) Each entity allocates its four corresponding out-going pins.

\( \Rightarrow \) The shortest path between two entities will be computed from out-going pin to out-going pin.

\( \Rightarrow \) All identified vertices will be added to a list, in the sequence they have to be visited.

\( \Rightarrow \) ExplorViz draws lines from point to point in the sequence they are placed in the list.

For an overview of all steps, see Figure 4.3.
4.4. Steps

Figure 4.3. All steps our approach takes until a layouting is done
Entity Placement

ExplorViz currently uses the rectangle-packing algorithm by [Wettel 2010] to place all entities on the drawing board. This layouting algorithm optimizes the space usage for placed objects. Due to the placement order depending on the size of each entity, neither this algorithm is stable, nor the edges can be laid out efficiently. Furthermore, the rectangle-packing algorithm in ExplorViz does not support saving of used and unused spaces. As a consequence, all free space between placed entities is unknown. A region search to locate collisions between edges and entities would consist of a loop which always iterates through all entities. Thus, ExplorViz would become unresponsive this way.

One goal of this thesis is a different approach for entity placement. Our approach was already given in Chapter 4. In this thesis we will use a quadtree and try to make it as stable as possible. Quadtrees allow saving of used and unused spaces, assure a faster search for collisions and help to install a grid structure for the edge layouting.

5.1 Placement Algorithm

The quadtree is defined in Definition 4.1. Its naturally given hierarchical structure fits to the purpose of building a stable and hierarchical 3D city layout in ExplorViz. As in Section 4.4 mentioned, the components and class'es of an application are saved in a tree structure. When switching from 2D landscape view to the 3D view of an application, this placement algorithm will be responsible for the appearance of all entities on a drawing board. Hence, the application indicates the whole drawing board. It is the foundation component, where all child entities will be placed on. All further components represent a new drawing board, respectively a new quadtree, where their children will be placed on. This hierarchy has to be displayed quite clearly.

To distinguish components from class'es, we capitalize on characteristics both differ from each other. Class'es are leaves of the component tree and since they do not contain any children, they will all be in size equal. The other way round, components are always nodes, but never leaves. They hold children and need to place them. Thus, each component will be a new drawing board, respectively quadtree, and got at least the size to enclose its children. The quadtree itself can be, due to the often heterogeneous size of components, far too large and exists of more unused than used space. To remedy this space mismanagement, all empty nodes of quadtrees will be cut off as far as possible.
5. Entity Placement

5.1.1 Calculation of Package Sizes

Since a component is like a package that exists of clazzes and/or further components, the size of a component depends on its children. In different words, the contained clazzes of a component dictate its size. Therefore, all clazzes get a fixed size first. Afterwards, the component sizes will be determined by recursively iterating through the component tree and calculating their size based on the children’s sizes (Algorithm 1). Prescribed by the regulations of quadtrees in Definition 4.1, each node has either four, in size equivalent, children, or contains an entity (Figure 4.1b).

The idea, based on [Eschbach 2008], is to put all entities in disjoint partitions in the quadtree as space efficient as possible, considering the mentioned definition. Apart from clazzes, all entities at the drawing board can be of different sizes. Thus, it appears that the smallest block, respectively subquadtree, has to include minimum the size of the smallest entity. As a consequence, the side lengths of all further nodes is a multiple of $2^n, n \in \mathbb{N}$, of the smallest entity’s side length. A side length of the smallest occurring block can be calculated by either raising the size of the largest, or the smallest entity, to an higher power. In course of the heterogeneous number and size of children, the smallest entity will be chosen to calculate the side length. The difference between both options can be seen in Figure 5.1.

![Figure 5.1. Difference in size calculation of quadtrees](image)

(a) Calculate side length by smallest entity  
(b) Calculate size length by biggest entity

To achieve an efficient and as quadratic as possible space usage at the drawing board, not the width and depth of a component’s children will be summed up to calculate a quadtree’s size. Instead, we sum up the base areas of all entities and compute a square’s area with side length of the smallest entity multiplied with $2^n, n \in \mathbb{N}$. This square is of about the same size and ensures the smallest side length fits $2^n, n \in \mathbb{N}$ times inside. Later this ensures all entities can be put on the quadtree and all child nodes include the same area.

Considering the characteristics of components the following three scenarios in size calculation have to be covered:
5.1. Placement Algorithm

(a) All entities fit into the quadtree, but the largest entity does not fit into a child node

(b) All entities fit into the quadtree

Figure 5.2. Ensure all entities fit into separate child nodes, if there are more than one entity to place

- Leaf nodes of the component tree only contain classes with fixed size, but no further components. Since all classes are equal in size, they are evenly distributed at four child nodes of a quadtree. The quadtree’s side length should match the square root of the number of classes to the next larger power of 2, multiplied with the fixed size of classes.

- The common case is a component that consists of further components and classes. How to calculate the size of the corresponding quadtree, was already described in this section. The base areas of all entities will be summed up and a square’s area with about the same size computed. Its side lengths will be the side length smallest entity multiplied with $2^n$, $n \in \mathbb{N}$. Under certain circumstances this approach leads to errors during entity insertion. Thus, the the next, worst, case has to be considered.

- In the worst case a component, with more than one child, creates a quadtree with an area $\text{smallestEntity.sizeLength} \times 2^d, d \in \mathbb{N}$, where all entities can fit in, but the covered area of the largest child node is less than the size of the largest component (Figure 5.2a). We bypass this by assuring the quadtree has at least a side length twice the size of the largest child’s side length, if a component has more than one child (see Figure 5.2b).

To avoid being out of space when trying to connect entities by edges, an inset space has to be added to all side lengths. Thus, all entities will have a small padding to the border of their surrounding quadtree. In Algorithm 2 this inset spaces is included.

5.1.2 Insertion

The purpose of the quadtree is to determine the positions of entities on the drawing board. The size of all entities and their corresponding quadtrees was calculated already. Our used
5. Entity Placement

Insertion algorithm is mostly taken from [Eschbach 2008]. To manage used and unused space as efficient as possible, each entity gets its individual node in a quadtree. Considering that, we have to pay attention to the node finding. All side lengths are of a power of 2 of the smallest contained entity’s side length. In addition, all child nodes on the same level are equal in size. Thus, tailored nodes for all entities can be computed. Since a quadtree’s nodes include a smaller area the deeper it is nested, the level, an entity has to be placed on, can be determined. For this the area all nodes one level deeper cover, can be calculated by dividing the quadtree’s base area by four. This step can be repeated until there is no smaller node an entity could be placed in. The number of divisions until the algorithm aborts, is the level, respectively the depth, an entity has to be placed. An example is shown in Figure 5.3.

Insertion orders influence the whole appearance of ExplorViz’s 3D view. Since the quadtree structure is remindful of quadrants in the Cartesian coordinate system, a natural approach would be starting in the first quadrant and insert in counterclockwise direction. This causes unnecessary expense when cutting quadtrees in Section 5.1.3. As a consequence, the origin will be chosen as the top left corner. Like in Figure 5.4 entities will be insert in clockwise direction. Entities can be inserted into a quadtree if and only if they meet the following conditions:

⇒ No entity is inserted already.

⇒ The quadtree, an entity shall be placed in, is a leaf.

⇒ The quadtree’s level is equal to the level an entity shall be placed on.

![Figure 5.3](image)

Figure 5.3. Given is a $16 \times 16$ large quadtree and a $2 \times 2$ large entity shall be inserted. Therefore, the $16 \times 16$ large field will be divided by 4 to get four, in size equivalent, nodes. Since the $8 \times 8$ large field is still larger than the $2 \times 2$ large field, it will be divide by 4 again. These step we repeat until a node is found that would became after a further division by 4 smaller than $2 \times 2$. In this example the level would be 3.
5.1. Placement Algorithm

To check, if a quadtree is a leaf on the level an entity shall be placed on, is important to avoid overlapping entities. Quadtrees always start with a depth 0. Its nodes will be splitted in four, in size equal, nodes, if the determined placement level of an entity that shall be placed, is higher than the level of the selected quadtree. As a consequence, if a quadtree has child nodes, it must have children in a higher depth and is no leaf. The gentle reader may notice, that the algorithm fills a node as far as possible before the next one will be filled. Hence, these quadtrees are in most cases profoundly unbalanced. On this we capitalize in Section 5.1.3, when quadtrees will be cutted in order to reduce their size. Especially, to eliminate as much unused space as possible.

A quadtree, like described in this section, can be seen in Figure 5.5. The unused space is far too large and has to be reduce to fit to the requirements. Compared to ExplorViz’s current placement algorithm we recognize that the segment, respectively component, size of the current layout says something of the number of entities it contains. By contrast, the
5. Entity Placement

The quadtree can become extremely large due to an unfortunate constellation of entities.

5.1.3 Cutting

At first glance on Figure 5.5, the quadtree does not look practical for usage in ExplorViz, since the unused space is far too large. After extensive look into the structure of the insertion algorithm in Algorithm 3, it is possible to detect a pattern the quadtree follows. Either all nodes except NW are empty, or the nodes SW, SO, or only SW (Figure 5.6). Empty nodes are a result of the size calculation that was did before. They evolve, if a large entity is followed by only a few small entities. This was discussed during size calculation in activity two, case three.

![Diagram showing different configurations of empty nodes](image)

*Figure 5.6. Pattern of which nodes could be empty, grey nodes are filled or partly filled with entities*

Following this pattern, all empty nodes can be cut off, without exceptions (see Figure 5.7a). A trivial approach could be deleting the whole node by setting it to `null`, but then all quadtree algorithms always would have to check all four nodes for `null` values before accessing them, in order to avoid `NullPointerException`. This causes changes in all procedures. Instead, the corresponding node’s width and depth will be set to 0 and a flag change to, is the quadtree cutted or not. Afterwards, the quadtree’s new size is the sum of widths and depths of its remaining nodes. When accessing a node, i.e. to create pins, the algorithm just checks, if a quadtree is cutted or not. No fatal error occurs, if the flag is not considered.

By virtue of its hierarchal setup, quadtrees in higher depths can be cutted to small sizes. Since the node is still filled, the size of its corresponding parent quadtree does not change (see Figure 5.7b). Accordingly, does the total size of the parent quadtree not shrink. Under the following conditions the unused space between two nodes (Figure 5.7b) can be squeezed:

- In vertical dimension unused spaces between two nodes can be removed, if the node SW is cut off or its content does not fill the whole node’s width.
- In horizontal dimension unused spaces between two nodes can be removed, if the node NO is cut off or its content does not fill the whole node’s depth.
5.1. Placement Algorithm

In this context does *squeeze* mean to adjust the parent quadtree size to a child node’s size. By the size a parent quadtree shrinks, the neighbor nodes’ coordinates, to the vertical or horizontal, have to be moved towards the origin. Theoretically, it is possible to cut a quadtree and move the neighbor nodes without exception. Nevertheless, this blasts the squareness of the quadtree and also the whole grid that is meant to be used later (see Figure 5.8a). Hence, the maximum space that can be cut off has to be identified. More precisely, either the maximum depth (*maxDepth*) of NW and NO, in order to cut in horizontal dimension, or the maximum width (*maxWidth*) of NW and SW, in order to cut in vertical dimension, has to be looked up. After cutting in vertical dimension, the nodes NO and SO will be moved by *maxWidth* towards the origin. If cutting in horizontal dimensions, the nodes SW and SO will be moved by *maxDepth* towards the origin. An example for quadtrees, before and after cutting off empty nodes, can be seen in Figure 5.7b and 5.8b.

![Figure 5.7](image)

*(a) Cutting off empty quadtree nodes (b) Possible quadtree result after cutting in higher depths*

**Figure 5.7.** Cutting off empty quadtree nodes. The red lines mark the new borders a the corresponding quadtrees.

At this point, it is apparent why using an insertion order with origin in the top left corner is a better decision. When cutting off unused space spaces, only two quadtree nodes have to be moved. Otherwise, it would have been possible all four nodes have to be moved, due to empty nodes at the left border. How cutting off empty nodes can reduce
5. Entity Placement

the quadtree’s size severely can be seen in Figure 5.9.

![Example of EPrints, using cutted quadtree as placement algorithm](image)

**Figure 5.9.** Example of EPrints, using cutted quadtree as placement algorithm

5.2 Stability

A new placement layouting approach was intended to be a stable alternative to the currently used layouting algorithm. Like [Steinbrückner 2012], we use an incremental approach for stability. An advantage of quadtrees over the rectangle-packing algorithm by [Wettel 2010], is the independence of entity positions concerning their size. This reduces entity relocations, especially if new components with, i.e. average size, evolve. While all entities have to be sorted by size to be placed by the rectangle-packing algorithm, a quadtree can place entities independent from their size. Placement sequences can be designed more efficient, i.e. to minimize edge crossings.

Our idea is based on the gradual evolution of cities. In the process of these cities’ development, often fixed structures, like city centers or suburbs, evolved. Fixed structures can have a high recognition value. Hence, the effort that is needed to build a mental map (Section 3.3), could be reduced. Since our quadtree’s insertion procedure is independent from entity sizes, a quadtree could grow without reordering all its entities. Consequently, two criteria are important:

- In two consecutive layoutings, all entities are inserted in the same sequence.
5.2. Stability

▷ If the quadtree growth between two consecutive layoutings, it grows by a factor of \(2^n, n \in \mathbb{N}\). This ensures the latest quadtree will be a node of a new, larger, quadtree.

ExplorViz’s current model structure (Figure 4.2) is not designed for stable layoutings. A component contains further components or clazzes, but neither does it save the sequence in that its children were inserted, nor does it save its latest size and position. Thus, we have to adapt the Component class to our requirements. Analogical, the Clazz object has to be adopted to save the latest size and position, too.

5.2.1 Preparation

Before adjusting our algorithms to be more stable, we adapt the structure of components and clazzes to our needs.

▷ To save the sequence in which all entities were inserted into the quadtree, add a list will be added to Component. In this approach called \texttt{insertionOrderList}. This list contains \texttt{Draw3DNodeEntity} objects, since components and clazzes both inherit from this class. New entities will be added at the end.

▷ All entities save their latest bounds, in this approach called \texttt{oldBounds}. That means, they backup their latest calculated size and also their coordinates on the drawing board. Saving insertion sequences of entities ensures all further layoutings, after an initial layouting, are independent from the order components and clazzes are saved in a parent component. While old coordinates only help to detect displacements between layoutings, the last calculated size of components is a crucial factor for the entire stability. If, i.e. the largest component, was removed from the component tree, the quadtree could loose in size and the placement could look quite different. Also, a new component could be added and a quadtree grows in size. However, not as much as it should be to ensure all contained entities are placed the same way again (see Figure 5.10). Both scenarios are unacceptable for stability.

5.2.2 Stability Implementation

After saving the insertion sequence and the latest bounds of entities, some procedures have to be adjusted, to make the layouting more stable. Therefore, the latest calculated size and \texttt{oldBounds} will be compared, before assigning a final size to an entity. Two scenarios have to be covered:

1. If the new calculated size of a component is less than before, one or more child components were removed after the previous layouting. The quadtree looses in size and hence could relocate entities on the drawing board. To avoid this, the new size of a component will be set to the previous one, backed in \texttt{oldBounds}.
5. Entity Placement

2. If the new calculated size of a component is larger than before, one or more child components were added after the previous layouting. The quadtree growth in size, but not large enough to ensure older entities will not be relocated. Therefore, the new side length of a component will be a factor of $2^n$, $n \in \mathbb{N}$ of its old side length, backed in oldBounds.

The second scenario is the more important one. As noticed in Section 5.1.2, the insertion algorithm fills up a node as far as possible, before filling up the remaining once. If a component’s quadtree did not change in size between two consecutive layoutings, although new entities were added, it still got space to assimilate them. In case it cannot assimilate this further entities, the component’s quadtree side length will be multiplied by $2^n$, $n \in \mathbb{N}$. As a consequence, the former quadtree becomes a node, more precisely the NW node, of a new quadtree and all entities will be placed exactly the same like before. However, entities that did not fit, will be placed in the next node (see Figure 5.10b). Therefor, the insertion sequence of entities are saved.

If a quadtree did not change in size, but an entity has to be added, the new entity will be placed in a remaining leaf, after all previous entities were placed like before. Same happens, if the quadtree grows in size. This steady growing allows fixed structures. The gentle reader may notices, that the quadtree never loses in size. Even if an entity was removed from a component, it still remains in insertionOrderList and will be placed in the quadtree. Since ExplorViz only draws entities, which are contained by children lists, a removed entity will leave a gap in the quadtree’s visualization, but still fills the leaf it filled before. Algorithm 4 shows the quadtree’s creation algorithm.

![Figure 5.10](image)

(a) A 8 × 8 large quadtree still got a free 2 × 2 large field  
(b) Double the size of a quadtree to insert a 4 × 4 large entity

**Figure 5.10.** When try to insert a 4 × 4 large entity into the 8 × 8 large quadtree in (a), the quadtree’s side lengths have to be double. This creates a new quadtree that contains the former one as NW node. The 4 × 4 large entity will be placed in the next node.
A central weak point of ExplorViz’s current Version is the edge layouting. The rectangle-packing layout by [Wettel 2010] takes only nodes but no edges into account. These leads to edges crossing entities. In [Wettel 2010] and [Eschbach 2008] are different approaches for layouting of edges presented. Both solutions do not fit to ExplorViz.

As in Figure 6.1 shown, does [Wettel 2010] use the three dimensional space for edges between visualization entities. Following all connections is almost impossible. This violates the requirements for ExplorViz and cannot be used as layouting solution.

At first glance the approach by [Eschbach 2008], shown in Figure 6.2, looks like it fits to ExplorViz, since the lines do not cross objects and are following an orthogonal embedding. The gentle reader notices, that edges can be bundled, if they follow the same pipe. Comparing this to the requirements in Section 4.3, this approach is ruled out, too. ExplorViz allows bundling of edges, if and only if two or more entities communicate with the same target.

A simple approach for layouting of edges based on [Eschbach 2008], will be described in detail in the following section. Although it fulfills most of the requirements, it cannot be used for the layouting of edges, since its performance is too bad. The approach for layouting edges that avoid crossing visualization entities, is creating a graph, which contains a grid with all possible connections. This grid is build on the borders of quadtrees. A shortest path between two entities can be determined by Dijkstra’s-Shortest-Path (DSP) algorithm,
6. Layouting of Edges

Figure 6.2. Approach by [Eschbach 2008] using pipes for connections

defined in Definition 3.2. The standard DSP algorithm computes all distances to a given target point. Using pins like in 6.1.1 slows the algorithm down. A priority queue, which manages all remaining pins, reduces the pins the algorithm visits to find a shortest path.

6.1 Edge Layouting Algorithm

6.1.1 Pin Creation

[Purchase et al. 2001] indicates orthogonal layouts as one good visualization possibility to establish a better understanding of connections and interactions. Consequently, a data structure is required that invariably allows horizontal and vertical edges. As explained before, a graph can meet this, if all vertices follow a grid structure. The points, representing graph vertices, are three dimensional coordinates at the border of all quadtrees and will be called pins in this thesis. A quadtree has up to nine pins, eight at the border (Figure 6.3a) and if it has children, one pin is set in the center (Figure 6.3b). For better understanding, the pins are called after the coordinates they possess on quadtrees: \{TopLeftCorner, NorthPin, TopRightCorner, EastPin, BottomRightCorner, SouthPin, BottomLeftCorner, WestPin, CenterPin\}.

Since placed entities got a padding to their surrounding quadtree, all entities allocate four pins, \{NP, OP, SP, WP\}, of their surrounding quadtree in order to use the quadtree’s border for edges. These pins will be called out – going pins in the following. In particular components need four out-going pins, although they create a quadtree itself. They have to be connected with the surrounding quadtree to make connections between entities on
6.1. Edge Layouting Algorithm

![Figure 6.3. Create pins on quadtrees](image)

Different hierarchy levels possible. Metaphorically, these out-going pins are the gates all traces pass to leave or enter an entity.

To enhance the performance of the path finding algorithm, cutted quadtree nodes will not create any pins.

### 6.1.2 Paths Building

After creating all pins, the graph’s edges have to be identified. For this purpose, all pins will be connected to their orthogonal neighbors, e.g. {TLC, NP} and {TLC, WP}. If all pins are connected, a square like in Figure 6.3a arises. In case a quadtree has children, the CP has to be connected with its orthogonal neighbors {NP, OP, SP, WP}, too (see Figure 6.3b).

Depending on if CP is set, 8-12 edges will build a grid for each quadtree. Considering the number of edges for this approach, an undirected graph is recommendable, since edges like {NP, TLC} are not needed to be created. Otherwise the number of edges is doubled and the performance gets more worse.

At same hierarchy level all possible paths are created now and all entities inside the corresponding quadtree can be connected. If components and classes communicate with entities in various components, they switch between hierarchy levels. Thus, crossover edges have to be installed by allocating the component’s out-going pins to the component’s quadtree pins (Figure 6.3c). A component allocates four out-going pins and creates a quadtree with at least eight pins. Be {CNP, COP, CSP, CWP} a component’s set of out-going pins and be the set of pins in Section 6.1.1, the pins of that component’s quadtree. Crossovers will be provided by creating the edges {{CNP, NP}, {COP, OP}, {CSP, SP}, {CWP, WP}}. Hence, a component has four crossings and looks like in Figure 6.4.

This is the minimal set of pins and edges that are needed, to allow a structured layouting of edges that suffices our requirements. The crossovers between hierarchy levels are a bottleneck and to prevent them, more pins for crossings could be created.
6. Layouting of Edges

![Figure 6.4. Creating crossovers between a component’s quadtree and its corresponding surrounding quadtree](image)

6.1.3 Finding Shortest Paths

As above mentioned, the standard DSP’s performance is not fast enough for layouting edges in ExplorViz. Thus, a priority queue with a depth first search is used. This way, the number of vertices the algorithm visits for finding and calculating the shortest path, is reduced. The priority queue uses a special comparator that first orders its queue by the distance to an target and secondly orders it by the direction the algorithm should use, if two vertices got the same distance to a target. Using a break-condition, if the algorithm arrives at the target pin first time, increases the performance of DSP, additionally.

Dijkstra’s shortest path algorithm needs a source and a target vertex. Each displayed trace in ExplorViz is an individual object, which saves a list of points that will be connected by lines, in the order they are saved. Based on this, we only need to determine all pins that build a path between two entities. Afterwards, they can be placed in the list of pins, in the sequence they shall be connected, respectively drawn. Currently, this list consist of two points, the center points of a source entity and a target entity. These two center points continue to be start and end points. In favour of the path finding algorithm, two different pins will be selected as source and target vertices. Hereinafter, it becomes apparent why all entities allocated four out-going pins. Center points of entities are no pins at the border of surrounding quadtrees. Thus, it is not ensured they are orthogonal embedded. For this reason, entities allocate orthogonal outgoing-pins at the surrounding quadtree’s border and ExplorViz draws a line between an entity’s center point and a selected out-going pin. Hence, a shortest path will be computed between two out-going pins. The identified pins that represent the path will be saved between start and end point.

These out-going pins have to be well selected. In this approach we calculate the Euclidean distance between the source entity’s out-going pins and the target entity’s out-going pins. An extract of how to determine two out-going pins, is shown in Figure 6.5a. Actually, NP, WP, OP would calculate their distance to the target’s out-going pins, too. The two out-going pins with the shortest Euclidean distance between, will be selected as source and target vertex for DSP. After DSP computed the shortest path, the designated points will be added to the trace’s point list. Afterwards, ExplorViz draws the path. An example
6.2 Stability

A stable edge layouting can be based on the idea in Section 5.2. After an initial layouting, all communications have to be saved in a separate list, i.e. named `stableCommunicationList`, in the sequence they were layouted. All further layoutings use `stableCommunicationList`, to draw all traces in exact the same sequence again. New communications will be added at the end of `stableCommunicationList`. Removed traces are still available in `stableCommunicationList` and will be handle just like regular traces. They will not be drawn, due to the same reasons why removed entities will not be displayed. As a consequence, we do not need to remove traces from `stableCommunicationList`, since they help to stabilize the edge layouting.
6. Layouting of Edges

**Figure 6.6.** Example the Dummy application, using cutted quadtree as placement algorithm and with layouted edges
Chapter 7

Evaluation

The previous chapters portrayed a new layouting algorithm based on quadtrees for entity placement and a graph for edge layouting. Whether the algorithm is a good alternative to the rectangle-packing algorithm already existing in ExplorViz or not, has to be evaluated. The first is a performance evaluation that analyses, if our layouting algorithm is more efficient than the current algorithm. Afterwards, both layouting algorithms will be evaluated concerning their stability. Each evaluation is based on different applications ExplorViz can visualize.

7.1 Performance Evaluation

Algorithm performance is a crucial factor for software engineers who use ExplorViz for reverse engineering, since this tool is meant to visualize a software landscape and its contained applications live. For this reason, layoutings should be generated efficiently and in short time. Otherwise, the claim being a tool for live trace visualization and simple reverse engineering is counteracted by long delays.

The term *performance* implies more than a pure runtime performance in this evaluation. Additional factors, like lucidity and quality of a visualization, influence the effort which an user has to invest to comprehend interactions inside an application. New layouting approaches should aim to minimize the effort an user needs to build a mental map of an application. Minimal requirements that aim for a good visualization of a new layouting algorithm, were given in Section 4.3. Beside a runtime evaluation, the length of edges, the squareness, size, and compactness of layoutings of different applications will be evaluated. To classify our approach the mentioned aspects will be evaluated with our new approach and the currently used rectangle-packing algorithm.

7.1.1 Test Scenarios

Since the layouting algorithm consists of different parts, these parts will be evaluated one by one. All scenarios will be evaluated with three algorithms:

- Rectangle-packing algorithm (current used layouting algorithm),
- Cutted quadtree (our approach), and
7. Evaluation

Quadtree.

Two quadtree options are used because our cutted quadtree moves entities after after insertion, in contrary to the convention. To allow an objective classification, both options will be evaluated.

Package and class structures of different applications influence the quality of layoutings. Some applications can look organized and clearly with a particular layouting algorithm, but the same layouting algorithm can layout an other application almost unrecognizable. All scenarios will be executed with the following applications, which deffer in size and number of classes and packages:

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Clazzes</th>
<th>Number of Components</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy</td>
<td>37</td>
<td>20</td>
<td>57</td>
</tr>
<tr>
<td>Kieker</td>
<td>34</td>
<td>25</td>
<td>59</td>
</tr>
<tr>
<td>EPrints</td>
<td>185</td>
<td>36</td>
<td>221</td>
</tr>
<tr>
<td>Kirat</td>
<td>163</td>
<td>165</td>
<td>328</td>
</tr>
</tbody>
</table>

Table 7.1. Used applications

ExplorViz filters communications between not visible entities. Additionaly, are duplicate traces bundled to a thicker edge in the layout. Table 7.2 gives an overview of the number of communications our four applications contain and how many will be computed in the two scenarios that are described in the following.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Communications Scenario 1</th>
<th>Communications Scenario 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy</td>
<td>5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Kieker</td>
<td>7</td>
<td>110</td>
<td>270</td>
</tr>
<tr>
<td>EPrints</td>
<td>169</td>
<td>577</td>
<td>1 203</td>
</tr>
<tr>
<td>Kirat</td>
<td>40</td>
<td>197</td>
<td>522</td>
</tr>
</tbody>
</table>

Table 7.2. Number of Communications
7.1. Performance Evaluation

S1: Performance Analysis on First Appearance

When switching from the 2D landscape view to 3D model view only the first level of components is opened. This configuration will be our starting point for all performance benchmarks. ExplorViz filters communications between not visible entities. As a consequence, the number of shown traces is limited to the number of communications between visible entities. Thus, at this starting point all algorithms should perform at their best.

S2: Performance Analysis all Components Opened

ExplorViz filters communications between not visible entities. As a result, the number of communications to be layouted, increases with the number of shown, in our case opened, entities. If all components are opened, no communications are filtered anymore and all contained classes, components, and traces have to be computed and displayed. Each algorithm should perform at its worst in this case.

7.1.2 Methodology

For each layouting algorithm different parameter influence their performance. Since we do not want to evaluate under which circumstances each algorithm performs at its best, we cherry picked a bunch of criteria with regard to their influence on the quality of layoutings, given in Section 4.3. In the following, these criteria will be described and their measurement defined. Since there are no benchmark tests for ExplorViz, the measuring functions will be implement inside all procedures that will be compared and evaluated.

Runtime

The runtime will be measured by taking the time all parts of the algorithm need to do their operations and print this time in a console. Measuring points are always before and after a step (see Figure 4.3).

\[
\text{runtime} = \text{endTime} - \text{startTime}
\]

Compactness

Compactness means that a layout uses its space as efficient as possible and unused space is minimized. A layouting is more compact than another, if it shows the same number of entities on less used space. To measure the compactness, the summed base areas of all children of the component tree’s root will be subtracted from the base area of the root component.

\[
\text{freeSpace} = \frac{\text{rootComponentBaseArea} - \sum_{\text{number of children}} \text{childArea}}{\text{number of children}}
\]

Afterwards, the free space will be divided by the number of entities the parent component contains. The less the average free space is, the more compact a layouting is. Hence, a
7. Evaluation

lower compactness coefficient is preferable.

Notably, this value represents the compactness coefficient of the fundament component. Due to the hierarchical setup of ExplorViz’s entity structure, all layouting algorithms insert entities the same way on all levels. If the first level is not compactly layouted, all further levels could still be compactly layouted. However, only by chance, due to a fortunate tree structure.

Squareness
Squareness will be measured as width in proportion to the depth of the root component. A value equal to 1 means the component is a square. If the value is less than 1, the depth is larger than the width. Vice versa, a value larger than 1 implies the width is larger than the depth. Following from this, the component is a rectangle.

\[ squareness = \frac{width}{depth} \]

Edge length
An edge consists of a list of pins. The whole length of an edge is the distance (Definition 3.4) between all points. Let \( n \) be the number of pins in the list, then the length of an edge is:

\[ length = \sum_{i=0}^{n-1} ||pin_{i+1} - pin_i|| \]

7.1.3 Experimental Setup

The evaluation will be conducted in the GWT developer mode in a Google Chrome browser. For each application both scenarios will be layouted 10 times by all three layouting algorithm. First we switch into the 3D model view of a corresponding application and let it layout 10 times. Thereafter, we use the "Open all Components" button to layout the same application 10 times with all components opened. All relevant data will be collected while layout generation and printed in the console. An overview of all steps is presented in Figure 4.3.

How all data will be collected:

- Runtime performance will be calculated by defining measuring points between all steps, one before and one after each step. The whole runtime is the time between the first and the last measuring point. For example, for edge layouting the follow measuring points will be set

1. Before and after pin creation
2. Before and after edge creation
3. Before and after path finding
7.1. Performance Evaluation

- The squareness coefficient will be the value \( \frac{\text{width}}{\text{depth}} \) from the root component.
- Free space will be determined by calculating the area of the root component and subtracting the areas of all its children.
- A layouting is compact, if it uses its space as efficient as possible. Thus, if the space between entities is reduced to the necessary minimum. Hence, the average free space is a compactness coefficient and will be calculated by dividing the free space by the number of children.
- For an average edge length all lengths will be summed up first and divided by the number of edges afterwards. The entire length, as well as the average length, will be collected.

7.1.4 Results

In this section the evaluation results are presented. An analysis of each application run will be presented in a separate subsection.

Different from the rectangle-packing algorithm, which determines each component size during entity insertion, quadtrees first calculate their size and insert all entities afterwards. Accordingly, our quadtree consists of more single parts than the rectangle-packing algorithm and will have more measuring points. To offer a realistic and objective opportunity for comparison, these additional measuring results will be summed up to a total value that can be compared to an equal part in the rectangle-packing algorithm.

Dummy Application

The Dummy application is only tailored for basic function tests in ExplorViz. It contains 57 entities, consisting of 37 classes and 20 components. Thus, it is the smallest application we use for our evaluation. All values in this section are averaged values.

Scenario 1  At the beginning the Dummy application has 57 entities to place and 5 communications to draw. First, we present results concerning the geometrical measurements in Table 7.3. The currently used layouting algorithm has a compactness coefficient of 67.65 and hence is more compact than both quadtrees options. Further, its edge length is of an average length of 57.67 and hence a third of the length of both quadtrees average edge lengths. With a squareness coefficient of 1, the quadtree is a perfect square. Our approach is, with a squareness coefficient of 2.48, more a rectangle than a square. An average runtime of each algorithm in Scenario 1 is shown in the following Table 7.4. The rectangle-packing algorithm is almost 6 times faster than our approach and more than 6 times faster than the standard quadtree implementation. Furthermore, the pure placement procedure of the rectangle-packing algorithm is 3 times faster than the quadtrees’ placement procedures. A layouting of edges is also computed 10 times faster by the current layouting algorithm.
7. Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>67.65</td>
<td>1.11</td>
<td>57.67</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>108.74</td>
<td>2.48</td>
<td>180.15</td>
</tr>
<tr>
<td>Quadtree</td>
<td>233.26</td>
<td>1.00</td>
<td>171.60</td>
</tr>
</tbody>
</table>

Table 7.3. Dummy’s average geometrical measurements in Scenario 1

than by both quadtree options. While the difference between the placement routines of both quadtrees is in a range of 0.5 ms, the difference between the edge layouting routines is 10 ms. Both, the pipe and the edge computation of the standard quadtree takes longer than of the cutted quadtree.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>insertion</td>
<td>pins</td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>5.20 ms</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>17.40 ms</td>
<td>4.60 ms</td>
<td>12.80 ms</td>
</tr>
<tr>
<td>Quadtree</td>
<td>17.90 ms</td>
<td>4.80 ms</td>
<td>13.10 ms</td>
</tr>
</tbody>
</table>

Table 7.4. Dummy’s average runtime performance results in Scenario 1

Scenario 2 In Scenario 2 all components are opened and all three algorithms have to place 57 entities and 9 communications to draw. Since all entity sizes are calculated by all three layouting algorithms before or during, placement of the entities, there is no difference in compactness and squareness between Scenario 1 and Scenario 2. Merely the edge length in all three approaches decreased (see Table 7.5).
7.1. Performance Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>67.65</td>
<td>1.11</td>
<td>57.53</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>108.74</td>
<td>2.48</td>
<td>99.14</td>
</tr>
<tr>
<td>Quadtree</td>
<td>233.26</td>
<td>1.00</td>
<td>121.14</td>
</tr>
</tbody>
</table>

*Table 7.5. Dummy’s average values of the geometrical measurements in Scenario 2*

A large difference in the runtime performance of each algorithm is shown in the following Table 7.6. While both quadtree options are 4, respectively 9, times slower than in Scenario 1, the rectangle-packing algorithm is 1.9 ms faster than before. With an average time of 5.6 ms the current layouting algorithm is 179.9 ms faster than the cutted quadtree and 412.9 ms faster than the uncutted quadtree. The performance difference between both quadtree options is with 229.5 ms far larger than in Scenario 1, when less communications had to be computed. Since both entity placement procedures differ in a maximum less than 6 ms, almost the entire runtime is caused of the quadtree’s pipe and edge creation, with an average difference of 230 ms and 232.50 ms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size insertion</td>
<td>pins pipes edges</td>
<td></td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>3.10 ms</td>
<td>2.40 ms</td>
<td>5.60 ms</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>17.00 ms</td>
<td>160.50 ms</td>
<td>185.50 ms</td>
</tr>
<tr>
<td>Quadtree</td>
<td>12.30 ms</td>
<td>395.50 ms</td>
<td>418.00 ms</td>
</tr>
</tbody>
</table>

*Table 7.6. Dummy’s average runtime performance results in Scenario 2*
7. Evaluation

Kieker

Kieker is with 59 contained entities in size comparable to our Dummy application. Unlike the Dummy application, Kieker is a real world application. Its structure is not optimized for ExplorViz and is a first severe test.

Scenario 1  At the beginning Kieker consists of 59 entities and 7 communications to draw. Table 7.7 presents the geometrical results of all involved layouting algorithms. Unusal is the compactness coefficient of the uncutted quadtree compared to ExplorViz’s currently used layouting algorithm and our new approach. The compactness coefficient of the quadtree is more than 1000 times larger than the compactness coefficients of the rectangle-packing algorithm and the cutted quadtree. A smaller difference can be seen in the squareness coefficient, again the quadtree is a perfect square, while the cutted quadtree is a rectangle, due to a squareness coefficient of 3.99. Concerning the edge length, the rectangle-packing algorithm draws in average 3 times shorter edges. Table 7.8 lists all average values of our runtime performance tests. It shows that ExplorViz’s current layouting algorithm is again faster than both quadtree versions, i.e. 7 to 12 times faster. With a difference of 16.90 ms to our approach and 18 ms to the standard quadtree implementation, the size and placement computation is done much faster by the rectangle-packing algorithm. Furthermore, both quadtree options layout all contained edges 7 to 15 times slower than the rectangle-packing algorithm.

Scenario 2  In Scenario 2 all components are openend and all three algorithms have to place 59 entities and 110 communications to draw. Again the compactness and squareness coefficients did not change between both scenarios, but the average edge length of both quadtree options is almost halved in Scenario 2 (see Tabel 7.9). Scenario 2 reveals a large runtime performance difference to Scenario 1. While the placement procedures of all three algorithms are at a maximum of 4 ms slower than before, the average edge layouting time increased for our standard quadtree options, from 92.90 ms in Scenario 1 to 216.90 ms in Scenario 2. Again the edge layouting, respectively path finding, procedures have a large

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>180.27</td>
<td>1.50</td>
<td>69.12</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>239.46</td>
<td>3.99</td>
<td>177.43</td>
</tr>
<tr>
<td>Quadtree</td>
<td>31 453.76</td>
<td>1.00</td>
<td>246.63</td>
</tr>
</tbody>
</table>

Table 7.7. Kieker’s average values of the geometrical measurements in Scenario 1
7.1. Performance Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>insertion</td>
<td>pins</td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>2.90 ms</td>
<td>4.60 ms</td>
<td>-</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>19.70 ms</td>
<td>33.00 ms</td>
<td>2.70 ms</td>
</tr>
<tr>
<td>Quadtree</td>
<td>20.90 ms</td>
<td>71.80 ms</td>
<td>5.20 ms</td>
</tr>
</tbody>
</table>

Table 7.8. Kieker’s average runtime performance results in Scenario 1

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>180.27</td>
<td>1.50</td>
<td>71.22</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>239.46</td>
<td>3.99</td>
<td>99.58</td>
</tr>
<tr>
<td>Quadtree</td>
<td>31,453.76</td>
<td>1.00</td>
<td>125.80</td>
</tr>
</tbody>
</table>

Table 7.9. Kieker’s average values of the geometrical measurements in Scenario 2

share on this slow runtime. The edge layouting algorithm of our approach is in Scenario 2 3 times slower than in Scenario 1.

EPrints

EPrints is a open-source software used for sharing digital, in most cases scientific, publications. It consists of 185 clazzes and 36 components. Hence it is not the largest application used for this performance test. Nevertheless, with in total 1203 traces it contains more traces than the 3 other applications involved in this evaluation.
7. Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>insertion</td>
<td>pins</td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>2.70 ms</td>
<td>-</td>
<td>8.60 ms</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>17.30 ms</td>
<td>-</td>
<td>727.60 ms</td>
</tr>
<tr>
<td></td>
<td>1.00 ms</td>
<td>16.30 ms</td>
<td>2.70 ms</td>
</tr>
<tr>
<td></td>
<td>2.30 ms</td>
<td>14.00 ms</td>
<td>1.60 ms</td>
</tr>
</tbody>
</table>

Table 7.10. Kieker’s average runtime performance results in Scenario 2

**Scenario 1** At the beginning EPrints has 221 entities to place and 197 communications to draw. ExplorViz’s rectangle-packing algorithm is with a coefficient of 52.94 once more, more compact than both quadtree options (139.18, 17596.16). Furthermore, the difference between the standard quadtree’s compactness and the compactness of our approach is large, too. Except in our approach, both the standard quad tree and the rectangle-packing algorithm layout EPrints like a square. All three approaches got an average edge length between 106.30 and 150.61. In Table 7.12 average runtime performances of all three layouting algorithms in Scenario 1, using EPrints, are presented. The rectangle-packing algorithm performed with 35.30 ms as best. Both quadtree options took for one layouting

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>52.94</td>
<td>1.06</td>
<td>106.30</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>139.18</td>
<td>2.45</td>
<td>150.61</td>
</tr>
<tr>
<td>Quadtree</td>
<td>17596.16</td>
<td>1.0</td>
<td>135.52</td>
</tr>
</tbody>
</table>

Table 7.11. EPrints’ average values of the geometrical measurements in Scenario 1
7.1. Performance Evaluation

in average more than 1 second (1071.90 ms, 1361.30 ms). Since the placement procedure performed almost identical in both quadtree versions, the edge layouting procedures of both, the cutted quadtree and the standard quadtree, differ in average 1003.30 ms for the cutted quadtree and 1311.40 ms for the uncutted quadtree.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>insertion</td>
<td>pins</td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>15.80 ms</td>
<td>-</td>
<td>35.30 ms</td>
</tr>
<tr>
<td>Cuted quadtree</td>
<td>37.00 ms</td>
<td>3.50 ms</td>
<td>33.50 ms</td>
</tr>
<tr>
<td>Quadtree</td>
<td>39.60 ms</td>
<td>5.10 ms</td>
<td>34.50 ms</td>
</tr>
</tbody>
</table>

Table 7.12. EPrints’ average runtime performance results in Scenario 1

Scenario 2  In Scenario 2 all 36 components contained in Epritns are opened and all three algorithms have to place 221 entities and 635 communications to draw. Like before, the compactness and squareness coefficients did not change between both scenarios, but the average edge length did. Table 7.13 shows the standard quadtree got, in average, almost twice as long edges than in Scenario 1. An large difference to Scenario 1 can be observed in Tabel 7.14. ExplorViz layouts EPrints with its current layouting algorithm in average in 106.90 ms, if all components are opened. In contrast, both quadtree options take 44 499.80 ms, respectively 76 131 ms for layouting. That means in average 44 and 76 seconds for one layouting, although the placement time stayed quite constant compared to Scenario 1. Additionally, the edge length did not increase significant, too. Thus, the edge computation took 95% of the layouting’s time to layout EPrints.

Kirat

Kirat is the largest application that will be compared to evaluate the performances of all three layouting algorithms. Its special composition of clazzes (163) and components
7. Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>52.94</td>
<td>1.06</td>
<td>126.44</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>139.18</td>
<td>2.45</td>
<td>171.65</td>
</tr>
<tr>
<td>Quadtree</td>
<td>17596.16</td>
<td>1.0</td>
<td>250.03</td>
</tr>
</tbody>
</table>

Table 7.13. Eprint’ average values of the geometrical measurements in Scenario 2

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>insertion</td>
<td>pins</td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>7.40 ms</td>
<td>99.50 ms</td>
<td>-</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>37.10 ms</td>
<td>44 499.10 ms</td>
<td>1.10 ms</td>
</tr>
<tr>
<td></td>
<td>1.10 ms</td>
<td>36.00 ms</td>
<td>1.80 ms</td>
</tr>
<tr>
<td>Quadtree</td>
<td>36.30 ms</td>
<td>76 094.70 ms</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1.50 ms</td>
<td>34.80 ms</td>
<td>3.10 ms</td>
</tr>
</tbody>
</table>

Table 7.14. EPrints’ average runtime performance results in Scenario 2

(165) is a challenge for our layouting algorithms, since all different applications we use, contain more clazzes than components. A result of the component to clazz ratio, not all components contain a clazz. However, child components must contain a clazz on any level. Thus, most components contain exactly one clazz, which is deep nested.

Scenario 1 At the beginning Kirat contains 328 entities and all three algorithms have to draw 40 communications. In Table 7.15 measurements concerning the geometrical data are presented, again averaged values of all 10 iterations. Compared to a compactness coefficient of 150.35, ExplorViz’s rectangle-packing algorithm is 1215 times more compact
7.1. Performance Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>150.35</td>
<td>0.84</td>
<td>148.60</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>816.19</td>
<td>1.63</td>
<td>99.58</td>
</tr>
<tr>
<td>Quadtree</td>
<td>182 786.28</td>
<td>1.0</td>
<td>1 362.24</td>
</tr>
</tbody>
</table>

Table 7.15. Kirat’s average values of the geometrical measurements in Scenario 1

than the standard quadtree implementation and about 5 times more compact than of our approach. An huge difference concerning the average edge length also can be seen between the standard quadtree and both other approaches. Average runtime performance test results of each algorithm is presented in Table 7.16. With 18.60 ms ExplorViz’s currently used layouting algorithm is faster than both quadtree options (213.20 ms cutted quadtree, 418.00 ms standard quadtree). First time there is a difference of 10 ms between the entity placement procedure of our approach and the standard quadtree. Both quadtree options layout their edges 205.20 ms and 473.00 ms slower than the rectangle-packing algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>insertion</td>
<td>pins</td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>10.50 ms</td>
<td>-</td>
<td>8.00 ms</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>104.70 ms</td>
<td>6.60 ms</td>
<td>213.20 ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.16. Kirat’s average runtime performance results in Scenario 1

used layouting algorithm is faster than both quadtree options (213.20 ms cutted quadtree, 418.00 ms standard quadtree). First time there is a difference of 10 ms between the entity placement procedure of our approach and the standard quadtree. Both quadtree options layout their edges 205.20 ms and 473.00 ms slower than the rectangle-packing algorithm.
7. Evaluation

**Scenario 2** In Scenario 2 all components are opened and all three algorithms have to place 328 entities and 197 communications to draw. Again the compactness and squareness coefficients did not change between both scenarios, but the average edge length. ExplorViz’s rectangle-packing algorithm halved its average edge length and the standard quadtree even reduced it by a third, while our approaches got in average twice as long edges than in Scenario 1 (see Table 7.17). If all components are opened and all traces have to be computed, all three algorithms need longer to place entities and connect them by edges. Table 7.18 shows a significantly worse runtime performance, compared to Scenario 1. The rectangle-packing algorithm takes with 35.30 ms in average twice as long than before in Scenario 1. A runtime increase of more than 10.000 ms (10 seconds) experiences our approach, the standard quadtree even takes all in all 32.231.10 ms (32.231 seconds) for the whole layouting. In all three cases the edge layouting procedures are to blame for the bad runtime performance.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compactness</th>
<th>Squareness</th>
<th>Edge length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>150.35</td>
<td>0.84</td>
<td>74.35</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>816.19</td>
<td>1.63</td>
<td>183.23</td>
</tr>
<tr>
<td>Quadtree</td>
<td>182786.28</td>
<td>1.0</td>
<td>360.53</td>
</tr>
</tbody>
</table>

Table 7.17. Kirat’s average values of geometrical measurements in Scenario 2

7.1.5 Discussion

The performance evaluation reveals large performance differences between all three algorithms. Here we discuss the results of our evaluation.

**Runtime performance**

Software engineers, who use ExplorViz for reverse engineering, expect an easy to use and organized tool with response times in real time. Figure 7.1 points out runtime differences of all three algorithms in both scenarios. Two facts can be highlighted: one, the currently used rectangle-packing algorithm performed in both scenarios for all four applications far better than both, our approach and the standard quadtree; two, in Scenario 2, when all components are opened and all traces are visible, both quadtree options are unusable. The average time for layouting Scenario 2 in EPrints took more than 30 seconds in our approach and more than 1 minute for the standard quadtree (see Table 7.14). This is far worse than in Scenario 1 for EPrints (see Table 7.12).
### 7.1. Performance Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Placement time</th>
<th>Edge layouting time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size insertion pins pipes edges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rectangle-packing</td>
<td>10.40 ms</td>
<td>24.90 ms</td>
<td>35.30 ms</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>102.50 ms</td>
<td>10 555.20 ms</td>
<td>10 657.20 ms</td>
</tr>
<tr>
<td></td>
<td>2.10 ms</td>
<td>100.40 ms</td>
<td>1 112.80 ms</td>
</tr>
<tr>
<td></td>
<td>5.10 ms</td>
<td>1 112.80 ms</td>
<td>9 437.30 ms</td>
</tr>
<tr>
<td>Quadtree</td>
<td>85.70 ms</td>
<td>29 406.60 ms</td>
<td>32 213.10 ms</td>
</tr>
<tr>
<td></td>
<td>2.70 ms</td>
<td>83.30 ms</td>
<td>3 355.40 ms</td>
</tr>
<tr>
<td></td>
<td>10.50 ms</td>
<td>3 355.40 ms</td>
<td>2 6040.70 ms</td>
</tr>
</tbody>
</table>

|                      |                  |                     |            |
|                      |                  |                     |            |

Table 7.18. Kirat’s average runtime performance results in Scenario 2

Mainly two reasons can be responsible for this long runtimes. On the one hand, a large number of pins and pipes have to be created to build a grid and connect all entities. On the other hand, bugs in the corresponding creation methods can cause errors in all following path findings. One missing pin on a quadtree’s boarder could lead to a small way round, but does not influence the entire layouting. Broken bottleneck roads however can paralyze the path finding algorithm. How the algorithm creates pins, pipes and edges was described in Chapter 6. As addressed, crossings between two quadtrees, from a component to their surrounding quadtree, are such bottleneck roads. Each placed component creates a new quadtree and connects this quadtree with its allocated out-going pins. To find a shortest path between two entities Dijkstra’s shortest path algorithm is implemented, with little performance improves. This algorithm checks for the most promising paths from a start pin to a target pin. If there is no such path to that target pin, it visits all possible pins, before it interrupts. Since each quadtree creates at least 8 pins with 8 corresponding pipes and all child nodes are quadtrees, an application with 200 and more entities provides a lot of pins and pipes. EPrints for example, contains in our approach 1 816 pins and 2 764 pipes when evaluating with the uncutted quadtree. Hence, if two classes in different components or hierarchy levels communicate with each other, but the crossings to the target’s quadtree are missing, the DSP algorithm still visits hundred of pins and pipes, before it interrupts. Doing this for 577 traces, like in EPrints, can be an explanation of these results for the standard quadtree. The fact that our cutted quadtree performs in its edge layouting far better, backs this assumption. Merged nodes do neither create pins nor pipes.
7. Evaluation

As a consequence the cutted quadtree creates only 1,291 pins and 1,812 pipes for EPrints. Accordingly, the DSP algorithm interrupts faster.

Even if our edge layouting procedure has no bugs, the algorithm is still far to slow for usage in a final version of ExplorViz. Kieker consist of only 59 entities and less than 1,000 pins create the entire grid, but its performance is poor, too. Since the rectangle-packing algorithm by [Wettel 2010] only takes nodes but no edges into account, ExplorViz does not have a proper edge layouting algorithm right now. It connects center points of two communicating entities directly and thus, no algorithm for edge layouting can perform better in its runtime. Nevertheless, both approaches lead to a dead end and an alternative approach should be found.

In the following, the focus will be on the results concerning the runtime performances of the pure placement algorithms. Figure 7.2 shows only measured placement runtimes. These results are more comparable, but show once more that the rectangle-packing algorithm is still faster than our approach. Since all approaches have in common that component and clazz sizes are calculate while or during their insertion, there should be no difference in runtime between Scenario 1 and Scenario 2 by the same layouting algorithm. These differences are reducible to measurement errors.

While our approach and the standard quadtree need about the same runtime to layout the Dummy application, Kieker, and EPrints, the runtime for layouting Kirat of both quadtree options differ in both, Scenario 1 and Scenario 2. For Kirat the standard quadtree performed significant better than the cutted option, twice. Our approach layouts a standard quadtree first and cuts off empty nodes afterwards. Due to its hierarchal structure, these cuttings starts at the deepest level and the algorithm iterates backwards through the
7.1. Performance Evaluation

Component tree. If two neighbor nodes of the same quadtree are cut off, all neighbors of its corresponding parent node can be affected and moved with all their contained entities. Since the movement algorithms recursively iterate to the deepest level again, quadtrees can be moved several times until the cut off procedure finishes. A structure with many components and an about one to one ratio between components and classes, like in Kirat, challenges this approach. Nevertheless, in worst case our approach took about 100 ms for a placement computation. This is still an acceptable response time.

Compactness

In order to place all entities in a separate node, the quadtree is at least four times as large as the largest child. How its size is calculated was explained in Chapter 5. A package structure with, e.g. deep nested classes, can cause larger components the closer they are to their root. If these components are followed by only a few small entities, several quadtree nodes could be left empty. In our approach we cut off these empty nodes, what makes it more compact than the standard quadtree in all tests. Nevertheless, the quadtree performed worst in Kirat. As mentioned before, Kirat’s component to class ratio is unfortunate for quadtrees. [Wettel 2010] described the rectangle-packing algorithm that is used in ExplorViz. Its entity arrangement is optimized for compact layoutings, due to its insertion order by size. Its compactness coefficient is on all hierarchy levels better than the quadtree’s. However, comparing EPrints’ layout in Figure 7.3 to our approach (Figure 5.9), the cutted quadtree layouted EPrints more clearly. On the other side, is Kieler
7. Evaluation

layouted beyond repair by our approach (Figure A.5) compared to a layouting with the rectangle-packing algorithm in Figure A.3. While we can reject the standard quadtree as new layouting algorithm, the cutted approach can look more organized than ExplorViz’s currently used placement algorithm.

Figure 7.3. Example of EPrints, using rectangle-packing algorithm as placement algorithm

**Squareness**

Concerning the squareness of a layouting, the standard quadtree got the best results. For all tested applications it layouted them as a perfect square (squareness coefficient of 1). In contrast, Figure 5.5 and its compactness coefficients show, this approach generates squareness on cost of space efficiency. All squareness coefficients of our cutted approach back this results, since in all layoutings both bottom nodes of the foundation quadtree were cut off. Hence, they were always empty. Best approach, concerning its compromise between both squareness and space efficiency, is the rectangle-packing algorithm that is already implemented in ExplorViz.

**Edge length**

According to the Pythagorean theorem, a line between two points is the shortest possible connection. As a consequence, it is not possible for any edge layouting algorithm to generate a shorter average edge length than the direct connections ExplorViz already
7.2 Stability Evaluation

implemented. Only a more compact layout or an optimized entity arrangement could impact positively. Especially, entity arrangement has a large influence on average sizes of edges. Since all entities are placed at the same spot in both quadtree options, the compression of unused space in the cutted quadtree leads to shorter edges in average, for almost all evaluated applications. An exception can be seen in Scenario 1 using EPrints. These average value of the standard quadtree can be reduced to errors in the path finding algorithm. As a consequence, most edges are direct lines, like in the rectangle-packing algorithm. A precisely analysis of this measurement cannot be made, due to bugs in the edge layouting procedures in our approach. Nevertheless, an advantage of our approach over the currently used edge layouting algorithm, can be seen in Figure 6.6.

7.1.6 Threats to Validity

To grade the quality of this performance evaluation, possible sources of errors have to be addressed. All tests were executed on only one computer. Furthermore, only four applications were involved in this evaluation and they have been layouted by all three algorithms. Following from this, we cannot ensure the performance results can be generalized for a fifth application.

More possible error sources could be bugs in our code, which never can be entirely excluded. In addition, operation system relevant background processes could have been active while single performance measurements and influence our results this way. Although, we turned off as many as processes as possible, Mac OS X still runs processes that cannot be turned off.

7.2 Stability Evaluation

The goal of this thesis is to implement a more stable 3D city layouting in ExplorViz. One state of an application can be layouted well. However, this is a static visualization and does not consider changes in the application’s structure, like the addition of new classes or packages. [Wettel 2010] gives an approach that considers changes and allows traveling in time in order to examine these changes. Since all states of an application are known before, this approach only feints stability.

ExplorViz’s intend is to allow reverse engineering live. Software landscapes and its contained applications can be updated anytime and hence it is not possible to know all states of an application before. Therefore, we implemented a new layouting algorithm that shall be more stable from scratch. Since the insertion procedure of the cutted quadtree is not tied to a fixed criteria, like an insertion sequence by size, newly evolved entities can be placed in any free node of a quadtree. In contrast, ExplorViz’s rectangle-packing algorithm has to place all entities with regard to their size. To test for a better stability than the rectangle-packing algorithm, we will evaluate both algorithms concerning their stability.
7. Evaluation

[Steinbrückner 2012] gives an overview of current similarity measures, which are based on empirical evaluation described in [Bridgeman and Tamassia 2002]. In this evaluation we will only concentrate on the number of movements, the average displacement of entities and the direction they were moved. As a consequence, we only evaluate if our approach moves less entities than the rectangle-packing algorithm used by ExplorViz. Furthermore, we do not evaluate the stability of the edge layouting.

7.2.1 Test Scenarios

Since our approach moves entities after empty nodes were cut off, we will evaluate again with the following three algorithms:

- Rectangle-packing algorithm (current used layouting algorithm),
- Cutted quadtree (our approach), and
- Quadtree.

All following scenarios will be conducted to our Dummy application and EPrints. Table 7.1 shows the number of entities contained in both applications.

**S1: Inserting aClazz to All Components**

Instead of inserting one single clazz at the lowest level, which would be inserted at the end by all three algorithms, we will test the scalability of the layoutings by adding a clazz to all components. Hence, each level grows in size and entities could be moved.

**S2: Inserting a Component With 10 Clazzes at First Level**

This scenario tests the stability of layoutings, if a new component with 10 clazzes is added on the first level of the component tree. Changes at the first hierarchy level can influence direct neighbors, but should not change the entire layouting.

**S3: Inserting 10 Clazzes at The Deepest Level**

Changes in the deepest level of the component tree can influence all previous levels. As a result, this is the most challenging scenario for all three algorithms in our evaluation.

7.2.2 Methodology

Each layouting approach has different characteristic, due to considered aspects others did not. Whereas the quadtree places entities into predefined areas, ExplorViz’s rectangle-packing algorithm calculates sizes and positions during the layouting process. [Steinbrückner 2012] described different similarity measurements. Instead of measuring the similarity
7.2. Stability Evaluation

between two consecutive layoutings, with regard to their relative positions before and after both layoutings, we limit this evaluation to the measurement of displacements. Therefore, we count the number of entities that were moved and additionally compute the average distance and the direction they were moved.

**Number of displacements**
If an entity was moved between two consecutive layoutings, its latest coordinates differ from the previous ones. The number of displaced entities compared to the first layouting will be counted.

**Average distance change**
A distance between two coordinates can be calculated by the Euclidean distance (Definition 3.4). To calculate the average distance, in which entities were moved, the difference between the positions in Scenario 1 and Scenario 2 of displaced entities will be measured and summed up. Afterwards, this total distance will be divided by the number of displacement.

**Average direction of movement**
How to calculate the angle of entities that was displaced is defined in Definition 3.5. The average direction entities were moved, is given by summing up all angles and dividing them by the number of displaced entities.

7.2.3 Experimental Setup

The evaluation will be conducted in the GWT developer mode in a Google Chrome browser. For both applications all three scenarios will be layouted by all three layouting algorithm. After a first layouting, the scenarios will be executed and entities created. In the following layouting all relevant data will be collected while layout generation and printed in the console.

**Where new entities will be added:**

**Dummy application**
- In Scenario 1 a new clazz will be added to each component.
- In Scenario 2 a component, consisting of 10 clazzes, will be added to the component "neo4j".
- In Scenario 3 10 clazzes will be add to the component "cache".
7. Evaluation

EPrints

大致在 Scenario 1 中一个新类将被添加到每个组件。

大致在 Scenario 2 中一个组件，包含 10 个类，将被添加到组件 "EPrints"。

大致在 Scenario 3 中 10 个类将被添加到组件 "Edit"。

在第二轮布局后，所有实体的坐标将与第一轮布局的坐标进行比较。如果它们有差异，移动计数器将增加。此外，这些坐标之间的距离也将测量，以及实体移动的方向。由于 ExplorViz 中的矩形堆叠算法不保存先前布局的实体位置，我们实现了它，就像在我们的方法中。

7.2.4 Results

在本节中，稳定性评估的结果将被呈现。每种应用程序运行的分析将被分别在一个单独的子节中。

Dummy Application

 Dummy 应用程序只用于基本功能测试在 ExplorViz。它包含 57 个实体，包含 37 个类和 20 个组件。因此，它是用于我们评估的最小的应用程序。

Scenario 1  在 Scenario 1 中，两种 quadtree 选项都移动了我们的 Dummy 应用程序包含的所有类。平均距离实体被切割 quadtree 移动的平均距离小于标准 quadtree。在切割 quadtree 和矩形堆叠算法中，所有实体主要在 X 轴上移动，远离原点。

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Displaced entities</th>
<th>Average distance</th>
<th>Average angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>25</td>
<td>58.00</td>
<td>187.46</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>37</td>
<td>13.69</td>
<td>188.68</td>
</tr>
<tr>
<td>Quadtree</td>
<td>37</td>
<td>120.80</td>
<td>159.40</td>
</tr>
</tbody>
</table>

Table 7.19. Results of the stability evaluation in Scenario 1 for the Dummy application
7.2. Stability Evaluation

**Scenario 2**  Tabel 7.20 presents the results of Scenario 2 for our Dummy application. Both the standard quadtree and our approach did not move any entities after the insertion of a new component with 10 clazzes on the first level of the component tree. On the contrary did the rectangle-packing algorithm used by ExplorViz, move 8 entities.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Displaced entities</th>
<th>Average distance</th>
<th>Average angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>8</td>
<td>86.42</td>
<td>211.97</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadtree</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.20. Results of the stability evaluation in Scenario 2 for the Dummy application

**Scenario 3**  While ExplorViz’s currently used layouting algorithm moved 16 entities in Scenario 3, both quadtree options moved only 2 entities. They moved those entities exactly the same and mostly on the X-axis, away form the origin (Tabel 7.21).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Displaced entities</th>
<th>Average distance</th>
<th>Average angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>16</td>
<td>50.42</td>
<td>133.60</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>2</td>
<td>47.66</td>
<td>184.84</td>
</tr>
<tr>
<td>Quadtree</td>
<td>2</td>
<td>47.66</td>
<td>184.8</td>
</tr>
</tbody>
</table>

Table 7.21. Results of the stability evaluation in Scenario 3 for the Dummy application

**EPrints**

EPrints consists of 185 clazzes and 36 components. Hence, it is the application with the highest number of clazzes we used in both evaluations.

**Scenario 1**  Similar to Scenario 1 for the Dummy application, both quadtrees moved all entities Eprints contains. The average directions they were moved differ in 6 degree. The rectangle-packing algorithm moved just 172 clazzes.
7. Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Displaced entities</th>
<th>Average distance</th>
<th>Average angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>172</td>
<td>43.68</td>
<td>119.33</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>185</td>
<td>115.12</td>
<td>136.74</td>
</tr>
<tr>
<td>Quadtree</td>
<td>185</td>
<td>159.89</td>
<td>130.81</td>
</tr>
</tbody>
</table>

Table 7.22: Results of the stability evaluation in Scenario 1 for EPrints

**Scenario 2** The insertion of a new component with 10 contained entities on the first level in the component tree does not cause any movements in our approach as well as in the standard quadtree (Table 7.23). In constrast, 57 entities were moved by ExplorViz’s current placement algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Displaced entities</th>
<th>Average distance</th>
<th>Average angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>57</td>
<td>69.15</td>
<td>123.99</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadtree</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.23: Results of the stability evaluation in Scenario 2 for EPrints

**Scenario 3** In Scenario 3 all involved algorithms moved all entities contained in EPrints. The rectangle-packing algorithm moved all entities mostly on the Z-axis, away from the origin (Table 7.24).

7.2.5 Discussion

All three scenarios for the Dummy application as well as Eprints showed that changes on deeper hierarchy levels influence the whole layouting. Both our approach and the rectangle-packing algorithm currently used in ExplorViz, moved in Scenario 1 and Scenario 3 for both application almost all entities. Since our approach differs in space usage from the rectangle-packing algorithm, the average distance between displaced entities and the average angular they were moved, cannot be compared to each other. Hence, these values only indicate a possible tendency where entities are moved by the approaches.
7.2. Stability Evaluation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Displaced entities</th>
<th>Average distance</th>
<th>Average angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle-packing</td>
<td>185</td>
<td>41.67</td>
<td>94.65</td>
</tr>
<tr>
<td>Cutted quadtree</td>
<td>185</td>
<td>115.129</td>
<td>141.10</td>
</tr>
<tr>
<td>Quadtree</td>
<td>185</td>
<td>161.71</td>
<td>128.09</td>
</tr>
</tbody>
</table>

Table 7.24. Results of the stability evaluation in Scenario 3 for EPrints

The results in Table 7.19 of Scenario 1 for the Dummy application show our approach moved all entities in almost the same direction. Furthermore, the average distance the entities were moved very short. This distance compared to the average distance the standard quadtree moved entities, points out how much unused space was cut off by the cutted quadtree. The average distance as well as average direction measure the absolute displacement only. Nevertheless, this does not give any evidence about the kind of displacement. If a layouting grows due to an evolution of the application, entities can be moved. Nevertheless, the relation to their neighbors can still be the same. Thus, a measurement of the absolute displacement of entities cannot proof the stability alone.

Scenario 2 points for both involved applications out that the placement procedure of the cutted quadtree, as well as the standard quadtree, does not depend on a insertion sequence of entities regarding to their sizes. As a consequence, newly evolved entities can be placed on any free space without causing a relocation of entities. However, only in the corresponding quadtree. Entities in different quadtrees can be affected, like shown in Scenario 3 for both, EPrints and the Dummy application.

7.2.6 Threats to Validity

Due to the lack of used measurements, we can not assure our approach is really stable. We did only show that our approach moves less entities in the two applications that were involved in this evaluation. Although, they differ both in size and in package structure, it is not ensured our approach moves less entities than the rectangle-packing algorithm in further applications. Furthermore, a growth of different package than we cherry picked, could have more influence to the whole layouting. Following, the depth of coverage is limited. Since the number of different scenarios, which could influence the layouting, is infinite, we focused on special characteristics of the rectangle-packing algorithm in ExplorViz and our quadtree approach.
Chapter 8

Related Work


One of the first 3D approaches to visualize the static architecture of software systems was proposed by [Knight and Munro 2000]. They motivated the usage of metaphors by quoting other researches in the information visualization field. Additionally, they introduce a concept called Software World that implements a real-world city metaphor. In this metaphor classes represent districts and methods the buildings in that districts. Packages and class attributes are not considered in this approach.

[Panas et al. 2003] proposed an approach that considered package structures of programming languages like Java. Packages were visualized as cities and classes as buildings. To increase realism, several elements with no semantic intend, such as trees, streets, and street lamps, were added. Furthermore, [Panas et al. 2003] allowed the first trace analysis by integrating dynamic data.

Due to the enhancement of computer performance and broadband internet, some researchers moved their software visualizations to the web. Web applications can increase accessibility to technology and reduce the inherent problems related to their installation and configuration. Such a 3D web application was proposed by [Mesnage and Lanza 2005]. The application visualizes software evolutions, based on the data stored in versioning system repositories.

The city metaphor has been studied in the EvoSpaces project [Lanza et al. 2009]. From that, two separate approaches evolved. One approach, in which software elements are represented by office buildings or city halls, was proposed by [Alam and Dugerdil 2007]. Several different layout strategies can layout cities, in which these elements are organized. Element references can be interactively displayed by animated solid pipes. These pipes also can show the direction of references.

[Wettel 2010] proposed the second approach called CodeCity. This approach visualized software decomposition and elements mapped onto city districts and buildings. The size of buildings is derived from basic size measures using a boxplot-based mapping technique presented in [Wettel and Lanza 2007]. Code Cities are meant as platform to support the identification of design problems [Wettel and Lanza 2008b]. Furthermore, it intends to support users to gain insight into the structural evolution of software systems [Wettel and Lanza 2008a]. A new idea of this approach is the possibility to step both forwards and
8. Related Work

backward in time, due to a time traveling technique that is build in. Thus, this approach allows visualizing the software system at its different evolution stages [Steinbrückner 2012]. For this, all versions of the software system are known in advance. Following from this, this approach only feints stability.

A more stable approach for a city metaphor is the EvoStreet approach by [Steinbrückner 2012]. An overview of this approach and additionally an evaluation of its stability, is given by [Tauer 2009].

Visualizing hierarchy and relationships emerged as a challenge for all researchers in the field of information visualization. Among the available techniques to visualize hierarchies, implicit visualizations and especially the Treemap dominate in approaches. These techniques’ advantage is a highly space-efficient graphical representation. Treemaps borrow from the map metaphor, hence they are, in general, used for 2D representation. [Shneiderman 1992] proposed a first approach for treemaps, in order to represent trees in a more efficient way. By that time, traditional trees were represented by, rooted, directed graphs. Their root node was at the top of the page and children nodes below the parent node with lines connecting them. However, treemaps represent hierarchically structured data, like trees are, as nested areas. Root nodes are, usually, represented as rectangles and child node rectangles completely cover their parental rectangle representation. [Shneiderman 1992] used in his algorithms data structures for spatial data representation [Samet 1984]. Due to the development in the field of information visualization, many treemap approaches have been proposed. [Schulz et al. 2011] give an overview of popular treemap data structures and describes them.
Conclusions and Future Work

In this bachelor’s thesis we proposed a new layouting approach for ExplorViz, based on quadtrees for entity placement and graphs for the edge layouting. The first, and standard, quadtree implementation wasted too much space in layoutings. Therefore, all empty nodes were cut off and the quadtree’s unused space compressed. In the performance evaluation the pure placement algorithm was less compact and was worse in its runtime performance than the currently used layouting algorithm. However, comparing EPrints’ layouted result of the cutted quadtree (Figure 5.9) to the rectangle-packing algorithm used in ExplorViz (Figure 7.3), shows that although this cutted quadtree is less compact, its layout can look more clearly. The advantage of our approach over the rectangle-packing algorithm is better consistency in its layouting. The stability evaluation showed our approach displaces less entities, if new entities evolve between two layoutings. The still acceptable placement runtime and its advantage of stability, makes it a possible alternative to the currently used rectangle-packing algorithm, but with restrictions. Not all applications are layouted well by our approach. Some applications got an unfortunate package structure for our cutted quadtree approach.

On the contrary, we found out that a grid structure managed by a graph is not an appropriate data structure for edge layouting in ExplorViz. Due to the performance evaluation, the graph showed an unacceptable lack of runtime performance. We improved Dijkstra’s shortest path algorithm by a priority queue, to visit the most promising pins only, in order to increase its performance. This improved algorithm was used during the performance evaluation. It is not advisable to hold on to the graph approach we implemented, since it performed bad for even small applications like the Dummy application. Instead, a new approach for the layouting of edges should be found.

Future Work

As future work the cutted quadtree could be improved and further evaluated concerning its stability. If a placed component grows too large and hence changes the node structure of a quadtree, the following entities are inserted to different nodes than before. Instead, it could be put at the end of the insertion list, in order to be placed in another node without influencing the position of following entities. Furthermore, an entity ordering procedure for the first layouting could be implemented. Since the quadtree is independent from
entity sizes in its insertion method, the insertion sequence lists of the entities could be
optimized concerning different aspects. By now, all entities are sorted by size, since we
only concentrated on the placement. Entities which are closely interrelated could be put
near by another. As a consequence, shorter edges are layouted. Additionally, an optimized
insertion sequence concerning the building of structures with a high recognition value is
possible, too.

A new approach for edge layouting has to be found. Since a new approach without
a suiting placement algorithm can be difficult, a completely different approach could
be tested. [Wang and Miyamoto 1996] present three different force-directed layouting
approaches. Force-directed layoutings are operating on graphs and allow to create graph
representations in an aesthetically pleasing way. Two forces, based on the physical behavior
of springs and particles, are mostly used by such algorithms. Hence, both attractive forces
as well as repulsing forces are influencing a layouting. A graph has a total energy that
is determined by vertices (particles) and edges (springs) between vertices. The layouting
algorithms configure a state when these forces are balanced. Since force-directed layoutings
concern both entity placement and edge layouting, this could be a solution for ExplorViz’s
layouting algorithm. Nevertheless, force-directed layoutings could have disadvantages
in runtime performance and stability. Due to the large amount of interactions between
the forces of vertices and edges, layouting algorithms can become very complex in order
to balance these forces. Furthermore, interactions can reduce the scaleability of the
corresponding layoutings and hence small structure changes can cause shiftings in the
whole graph representation.
Bibliography


Bibliography


This chapter presents the algorithms, which were described in this thesis.

Size Calculation

The size calculation consists of two parts. First Algorithm 1 sets a fixed size all clazzes. Afterwards, all component sizes will be calculated in Algorithm 2.

Algorithm 1 Apply Metrics to Clazzes

1: procedure APPLYMETRICS(Component component)
2:     for all childcomponent ∈ component.children do
3:         applyMetrics(childcomponent)
4:     end for
5:     for all clazz ∈ component.clazzes do
6:         clazz.width ← fixedSize
7:         clazz.depth ← fixedSize
8:     end for
9:     calculateSize(component)
10: end procedure
Algorithm 2 Calculate the Size of Components and Quadtrees

1: **procedure** calculateSize(Component component)
2:     size ← 0
3:     sideLength ← 0
4:     if component.children.empty = false then component.children.sortBySizeDesc
5:     end if
6:     for all child ∈ component.children do
7:         size ← size + area(child, insetSpace)
8:     end for
9:     for all clazz ∈ component.clazzes do
10:        size ← size + area(clazz, insetSpace)
11:     end for
12:     smallestElement ← smallest entity in component
13:     i ← 0
14:     found ← false
15:     while found = false do
16:         if size ≤ area(smallestElement, insetSpace) × area(pow(2, i)) then
17:             found ← true
18:         else
19:             i ← i + 1
20:         end if
21:     end while
22:     sideLength ← (smallestElement.width + insetSpace) × pow(2, i)
23:     if component.children.size > 1 then
24:         if sideLength/2f ≤ (component.firstChild.width + insetSpace) then
25:             factorLength ← component.firstChild.width + insetSpace
26:             if component.children.size = 4 ∧ component.clazzes.empty = false then
27:                 if 2 × component.lastChild.width ≥ factorLength then
28:                     factorLength ← component.lastChild.width + insetSpace
29:                 end if
30:             end if
31:         end if
32:         if first and last child component equal in width and depth then
33:             if component.clazzes.empty = false ∧
34:                 (log(component.children.size)/log(2))%2 = 0 then
35:                 factorLength ← component.firstChild.width + insetSpace
36:             end if
37:         end if
38:     end if
\[ \text{sideLength} = 2 \times (\text{sideLength} + \text{insetSpace}) \]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{else if component.children.size} = 1 \text{ then}
\]

\[
\text{if component.clazzes.empty} = \text{false then}
\]

\[
\text{if component.firstChild.width + insetSpace} \geq \text{sideLength}/2 \text{ then}
\]

\[
\text{sideLength} \leftarrow 2 \times (\text{smallestElement.width + insetSpace}) \times \\
\text{pow}(2, \log(\text{ceil}((\text{component.firstChild.width + insetSpace})/(\text{smallestElement.width + insetSpace}))/\log(2)))
\]

\[
\text{else}
\]

\[
\text{sideLength} \leftarrow \text{component.firstChild.width + insetSpace}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{else}
\]

\[
\text{requiredSpace} \leftarrow \text{ceil}(\sqrt{\text{component.clazz.size}})
\]

\[
\text{p} \leftarrow 0
\]

\[
\text{foundSpace} \leftarrow \text{false}
\]

\[
\text{while foundSpace} = \text{false do}
\]

\[
\text{if pow}(2, \text{p}) \geq \text{requiredSpace} \text{ then}
\]

\[
\text{foundSpace} \leftarrow \text{true}
\]

\[
\text{else}
\]

\[
\text{p} \leftarrow \text{p} + 1
\]

\[
\text{end if}
\]

\[
\text{end while}
\]

\[
\text{end if}
\]

\[
\text{if component's old bounds exist then}
\]

\[
\text{largestElement} \leftarrow \text{largest entity in component}
\]

\[
\text{if sideLength} > \text{component.oldBounds.width} \land \text{sideLength}/2 \leq \\
\text{largestElement.width + insetSpace} \text{ then}
\]

\[
\text{sideLength} \leftarrow 2 \times (\text{component.oldBounds.width + insetSpace})
\]

\[
\text{else if component.oldBounds.width} > \text{sideLength} \text{ then}
\]

\[
\text{sideLength} \leftarrow \text{components.oldBounds.width}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{component.width} \leftarrow \text{sideLength}
\]

\[
\text{component.depth} \leftarrow \text{sideLength}
\]

\[
\text{end procedure}
\]
. Algorithms

Insertion

This section presents the insertion algorithm of the quadtree used in this thesis.

Algorithm 3 Insert Entities into a Quadtree

1: procedure insert(quadtree quad, Draw3DNodeEntity entity)
2:   if quad is full then
3:     return false
4:   end if
5:   insertingDepth ← computeLevel(entity, boundsquadtree, quad.level)
6:   if insertingDepth = quad.level ∧ quad has children then
7:     return false
8:   end if
9:   if insertingDepth > quad.level then
10:      if quad has no children then
11:         quad.split()
12:      end if
13:      if insert(quad.NW, entity) = true then
14:         return true
15:      else if insert(quad.NO, entity) = true then
16:         return true
17:      else if insert(quad.SO, entity) = true then
18:         return true
19:      else if insert(quad.SW, entity) = true then
20:         return true
21:      else
22:         return false
23:      end if
24:   else
25:      if quad has children then
26:         return false
27:      end if
28:      quad.objects.add(entity)
29:      return true
30:   end if
31: end procedure
Creation

In this section the algorithm that creates all quadtrees is presented.

Algorithm 4 Create Quadtree

1: procedure CREATEQUADTREE(Component component)
2: quad ← newQuadtree(0, new Bounds(component.X, component.Y, component.Z,
3: component.width, component.height, component.depth))
4: if component’s insertion order list is empty then
5: component.insertionOrderList.addAll(component.children)
6: component.insertionOrderList.addAll(component.clazzes)
7: else
8: for all child ∈ component.children do
9: if component.insertionOrderList.contains(child) = false then
10: component.insertionOrderList.add(child)
11: end if
12: end for
13: for all clazz ∈ component.clazzes do
14: if component.insertionOrderList.contains(clazz) = false then
15: component.insertionOrderList.add(clazz)
16: end if
17: end for
18: end if
19: for all entity ∈ component.insertionOrderList do
20: inserted ← quad.insert(quad, entity)
21: if entity instanceof Component then
22: createQuadtree(entity)
23: end if
24: end for
25: if inserted = true then
26: component.putOldBounds
27: quad.cutOffNode(quad)
28: component.quadTree ← quad
29: component.adjust
30: end if
31: end procedure
Cutting

The cutting of quadtrees is an important part of this thesis. Therefore, the two algorithms, developed in this thesis, will be present in this section.

**Algorithm 5 Cut Quadtree**

```plaintext
1: procedure cutOffNode(Quadtree quad)
2:   if quad has nodes then
3:     cutOffNode(quad.NW)
4:     cutOffNode(quad.NO)
5:     cutOffNode(quad.SO)
6:     cutOffNode(quad.SW)
7:   end if
8:   if quad is empty then
9:     quad.width ← 0
10:    quad.depth ← 0
11:   end if
12: end procedure
```
Algorithm 6 Adjust Quadtree

1: procedure \textsc{adjustQuadtree}(Quadtree \textit{quad})
2: \hspace{1em} \texttt{cutX} $\leftarrow$ 0
3: \hspace{1em} if \textit{quad} has nodes then
4: \hspace{2em} \texttt{cutOffNode}\textit{quad}.NW
5: \hspace{2em} \texttt{cutOffNode}\textit{quad}.NO
6: \hspace{2em} \texttt{cutOffNode}\textit{quad}.SO
7: \hspace{2em} \texttt{cutOffNode}\textit{quad}.SW
8: \hspace{1em} if \textit{quad}.NW.cutted = true \wedge \textit{quad}.NO.cutted = true then
9: \hspace{2em} \textit{quad}.width $\leftarrow$ \textit{quad}.NW.width
10: \hspace{1em} else
11: \hspace{2em} \texttt{leftWidth} $\leftarrow$ 0
12: \hspace{2em} \texttt{rightWidth} $\leftarrow$ 0
13: \hspace{2em} if \textit{quad}.NO.width \geq \textit{quad}.SO.width then
14: \hspace{3em} \texttt{rightWidth} $\leftarrow$ \textit{quad}.NO.width
15: \hspace{2em} else
16: \hspace{3em} \texttt{rightWidth} $\leftarrow$ \textit{quad}.SO.width
17: \hspace{1em} end if
18: \hspace{1em} if \textit{quad}.NW.width \geq \textit{quad}.SW.width then
19: \hspace{2em} \texttt{leftWidth} $\leftarrow$ \textit{quad}.NW.width
20: \hspace{1em} else
21: \hspace{2em} \texttt{leftWidth} $\leftarrow$ \textit{quad}.SW.width
22: \hspace{1em} end if
23: \hspace{1em} if \textit{quad}.NW.X + \texttt{leftWidth} < \textit{quad}.NO.X then
24: \hspace{2em} \texttt{cutX} $\leftarrow$ \textit{quad}.NO.X $-$ (\textit{quad}.NW.X + \texttt{leftWidth})
25: \hspace{2em} \texttt{moveQuadtreeX}\textit{quad}.NO, -\texttt{cutX}
26: \hspace{2em} \texttt{moveQuadtreeX}\textit{quad}.SO, -\texttt{cutX}
27: \hspace{1em} end if
28: \hspace{1em} \textit{quad}.width $\leftarrow$ \texttt{leftWidth} + \texttt{rightWidth}
29: \hspace{1em} end if
30: \hspace{1em} if \textit{quad}.SW.cutted = true \wedge \textit{quad}.SO.cutted = true then
31: \hspace{2em} if \textit{quad}.NW.depth < \texttt{quad}.depth \wedge \textit{quad}.NW.depth \geq \textit{quad}.NO.depth then
32: \hspace{3em} \texttt{quad}.depth $\leftarrow$ \textit{quad}.NW.depth
33: \hspace{2em} else if \textit{quad}.nodes.NO.depth $<$ \texttt{quad}.depth \wedge \textit{quad}.NW.depth $<$ \textit{quad}.NO.depth then
34: \hspace{3em} \texttt{quad}.depth $\leftarrow$ \textit{quad}.NO.depth
35: \hspace{2em} end if
else

topDepth ← 0
bottomDepth ← 0
if quad.NW.depth ≥ quad.NO.depth then
    topDepth ← quad.NW.depth
else
    topDepth ← quad.NO.depth
end if
if quad.SO.depth ≥ quad.SW.depth then
    bottomDepth ← quad.SO.depth
else
    bottomDepth ← quad.SW.depth
end if
if quad.NW.Z + topDepth < quad.SW.Z then
    cutZ ← quad.SW.Z − (quad.NW.Z + topDepth)
    moveQuadtreeZ(quad.SO, −cutZ)
    moveQuadtreeZ(quad.SW, −cutZ)
end if
quad.depth ← topDepth + bottomDepth
end if
if quad contains entity then
    if quad.NW instanceof Component then
        marginLeft ← quad.NW.X − quad.X
        marginTop ← quad.NW.Z − quad.Z.
        quad.width ← quad.entity.width + 2 × marginLeft
        quad.depth ← quad.entity.depth + 2 × marginTop
    end if
end if
end if
end procedure
Additional Figures

In this chapter the layoutings of the Dummy application, Kieker, and Kirat are present.

Dummy Application

In the following Figure A.1 the Dummy application is presented. The layouting was done with the currently used layouting algorithm in ExplorViz.

![Dummy Application Layout](image)

**Figure A.1.** Example of the Dummy application, using rectangle-packing algorithm as placement algorithm
Kieker

This section presents layoutings of Kieker in Scenario 1 and Scenario 2 in Section 7.1. The following figures show Kieker layouted by both, the rectangle-packing algorithm and the cutted quadtree.

![Diagram of Kieker layout]

**Figure A.2.** Example of EPrints, using cutted quadtree as placement algorithm.
**Figure A.3.** Example of Kieker, using rectangle-packing algorithm as placement algorithm and all components are opened
. Additional Figures

Figure A.4. Example of Kieker, using cutted quadtree as placement algorithm.

Figure A.5. Example of Kieker, using cutted quadtree as placement algorithm and all components are opened
Kirat

This section presents layoutings of Kirat in Scenario 1 and Scenario 2 in Section 7.1. The following figures show Kirat layouted by both, the rectangle-packing algorithm and the cutted quadtree.

Figure A.6. Example of Kirat, using rectangle-packing algorithm as placement algorithm
Additional Figures

Figure A.7. Example of Kirat, using rectangle-packing algorithm as placement algorithm and all components are opened
Figure A.8. Example of Kirat, using cutted quadtree as placement algorithm.

Figure A.9. Example of Kirat, using cutted quadtree as placement algorithm and all components are opened.
Appendix

DVD Containing Created Source Code and Data

This chapter specifies all contained content on the DVD.

Content on the DVD

For a better overview, all content is placed in separate folders.

- **ExplorViz** contains the whole ExplorViz project (source code).
- **Images** contains all pictures that were used in presentations and this bachelor’s thesis.
- **PDF** contains the Proposal, all presentations and this bachelor’s thesis as PDF.