МИНОБРНАУКИ РОССИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»   
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

Применение генетического алгоритма для реализации трёхмерной укладки грузов в кузов перевозчика

Бакалаврская работа

02.03.02 Фундаментальная информатика и информационные технологии

Инженерия программного обеспечения

Зав. кафедрой д.т.н., профессор \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Абрамов Г. В.

Обучающийся Денисов А.Б.  *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Руководитель

к.т.н., доцент Авсеева О. В. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Воронеж 2022

**Оглавление**

|  |  |
| --- | --- |
| Список сокращений……………..……………………………………….. | 3 |
| Введение………………………………………………………………….. | 4 |
| 1. Алгоритм укладки...…..……………………………………………….. | 8 |
| 1.1. Обзор существующих решений задачи………………………….. | 8 |
| 1.2. Решение задачи на основе уровневых алгоритмов………….….. | 10 |
| 1.3. Генетический алгоритм…………………………………………... | 13 |
| 1.4. Решение задачи на основе генетического алгоритма………...… | 14 |
| 2. Web приложение «Укладка»……………..…..……………….………. | 7 |
| 2.1. Элементы приложения…………………………………………… |  |
| 2.2. Firebase и бизнес логика…………………………………………. |  |
| 2.2.1. Авторизация и аутентификация…… |  |
| 2.2.2. Журнал учёта укладок…… |  |
| 2.3. 3D – визуализация укладки…… |  |
| 2.4. Реализация алгоритмов в приложении………..……………… | 8 |
| Заключение……………………..………………………………………….. | 11 |
| Список использованных источников………… |  |
| Приложение………… |  |

**Список сокращений**

**JS** – JavaScript.

**JSON** – JavaScript Object Notation – формат хранения данных в виде JavaScript объектов.

**API** – Application Programming Interface – программный интерфейс приложения.

**JSP** – Java server pages.

**ООП** – объектно-ориентированная парадигма программирования.

**ФП** – функционально-ориентированная парадигма программирования.

**Введение**

В современной логистике в последнее время становится крайне актуальным вопрос оптимизации доставки грузов. Важнейшую роль в доставке играет способ укладки грузов в кузов машины, которая будет развозить грузы. Это обуславливается несколькими факторами. Во-первых, чем больше суммарное количество грузов, поместившихся в грузовик, тем меньше затраты на топливо, а в условиях загруженного городского трафика это особенно важно. Во-вторых, важен порядок укладки упаковок. Если нужная в данном пункте коробка будет в конце кузова, то для того чтобы её достать придётся потратить время на частичную разгрузку кузова – это не оптимально, следовательно, этого нужно избегать.

Вышеописанная задача может быть сведена к задаче трёхмерной укладки коробок (параллелепипедов) в заданный кузов (объём). Данная задача давно известна и глубоко исследована в математике **([1])** и является NP полной задачей (т.е. не имеет точного решения за полиномиальное решение). Исходя из этого практический смысл имеют эвристические подходы к решению данной задачи. Далее будет рассмотрено несколько эвристических подходов, они буду сравниваться и лучшие из них в отношении поставленной задачи будут использованы в реализации алгоритма.

Кроме математической составляющей поставленной задачи следует рассмотреть и её техническую сторону, то есть программную реализацию. Для реализации задачи для начала следует сравнить стеки современных технологий и выбрать подходящий набор.

Поскольку задача подразумевает реализацию алгоритма, для которого будет разработан некоторый интерфейс для передачи данных функции алгоритма и интерфейс для получения результата работы алгоритма, то становится ясно, что главная сложность в алгоритмизации задачи, а не в построении большой архитектуры приложения.

Сравним три стека технологий. Два из них похожи – это Java и .Net (C#). Раньше C# был ограниченным из-за отсутствия поддержки macOS и Linux-подобных систем, но с появлением преемника .Net Framework в лице .Net Core возникла возможность писать кроссплатформенные приложения. В свою очередь, Java является проверенным решением, с помощью которого можно реализовывать web-приложения как с помощью рендеринга на сервере, так и с совмещением с популярными фреймворками JavaScript. Хотя на C# возможна реализация вышеописанных вариантов, Java выигрывает в написании реализации мобильного приложения из-за Kotlin. В целом, обе технологии подходят для реализации задачи, но являются громоздкими и избыточными решениями, которые подразумевают классическую структуру приложения включающую реляционную базу данных и backend приложение. Кроме того, в последнее время монолитная структура web-приложения становится менее популярной (такие технологии как .Net MVC, JSP), а значит поддержка и сообщество у данных инструментов сокращается.

Как было сказано ранее, у приложения не будет сложной архитектуры, основные данные которые оно будет хранить – пользователи и результаты их укладок (т.е. простейшая статистика). Для данной задачи идеально подходит Firebase – решение от Google, представляющее собой NoSQL облачную базу данных и по сути готовый backend. Firebase часто используется для стартапов и простых приложений, не содержащих сложной бизнес логики, что как раз и совпадает с нашим случаем.

Для реализации основной части приложения предлагается использовать ставший уже классическим frontend фреймворк React.js. React позволяет динамически отрисовывать элементы страницы, в зависимости от данных бизнес логики (хуки useState, useEffect). Эта функция будет крайне удобна при визуализации укладки. В сравнении со своими прямыми конкурентами Angular.js и Vue.js, React имеет более лёгкую и настраиваемую структуру, в отличии от Angular, и большее сообщество, в сравнении с Vue.js. Также стоит отметить, что для React существует множество пакетов для интеграции с Firebase.

Для удобства пользователя и наглядности работы алгоритма имеет смысл разработки 3D визуализации укладки. Существует библиотека-обёртка над WebGL (аналог OpenGL, для веб-браузеров) под названием Three.js, которая предоставляет удобный интерфейс пользования для frontend разработки. Конкретно для React, Three.js содержит библиотеку-адаптер react-three-fiber, позволяющую пользоваться функциями Three.js в декларативном стиле компонентов React.

Конечной целью данной выпускной квалификационной работы является разработка алгоритма, осуществляющего вычисление оптимальной трёхмерной укладки грузов, и приложения, которое предоставляет интерфейс для введения данных укладки и получения результата работы алгоритма.

Конкретно при выполнении работы необходимо решить следующие задачи:

* Касательно алгоритма:
  + Рассмотреть основные эвристические решения задачи трёхмерной укладки;
  + На основе рассмотренных эвристических решений составить алгоритм оптимальный в рамках текущей задачи;
  + Оценить оптимальность укладки для реализованного алгоритма использую подсчёт объёмов;
  + Оценить оптимальность на выборке случайно сгенерированных данных.
* Касательно приложения требуется реализовать:
  + Дружелюбный интерфейс для ввода данных о контейнере и упаковках;
  + Экспорт данных результата работы алгоритма укладки в формате JSON;
  + Генератор случайных погрузок. То есть, должна происходить генерация некоторого количества контейнеров с их размерами (случайные числа должны быть ограничены диапазоном);
  + 3D просмотр укладки;
  + Две роли – гость и пользователь;
  + Результаты укладок должны сохранятся в общую таблицу для каждого пользователя.
* Также:
  + Освоить базовую функциональность Firebase;
  + Освоить базовую функциональность Three.js.

1. **Алгоритм укладки**

Как было сказано во введении данная задача имеет только решения в виде эвристических алгоритмов. Далее рассмотрим несколько вариантов таких алгоритмов.

* 1. ***Обзор существующих решений задачи трёхмерной укладки***

Достаточно часто в решении задачи трёхмерной укладки прибегают к далее описываемому подходу. Последовательно по определённым правилам строят слои, укладывая их либо друг на друга или же последовательно, пока не будет заполнена плоскость основания, а затем сверху этого слоя – то есть в виде стен или же плоских слоев. Стоит отметить, что один из этих вариантов укладки выбирается в зависимости от требований несущих способностей нижележащих слоев (в данной работе не используется). Одна из главных решаемых задач и оптимизационных параметров – это плотность укладки. Определим плотность укладки:

(1)

где Vi – объем i-ой упаковки, а – объем получившейся упаковки. Объём собранной упаковки будет всегда больше или равен сумме объёмов всех упаковок из-за того, что при упаковке образуются пустоты. В вышеописанных подходах перед укладкой осуществляется сортировка по высоте коробок или же по площади основания. Далее, при укладке каждого слоя задача сводится к двумерной укладке, для которой существует несколько подходов (чаще всего говорят про уровневые, шельфовые и плоские алгоритмы). Рассмотрим так называемые уровневые алгоритмы, в них сначала производится сортировка массива упаковок по какому-либо метрическому параметру.

Самый очевидный вариант - алгоритм Next Fit Decreasing Height (NFDH). Сначала упаковки упорядочиваются по невозрастанию высоты, затем производится укладка уровня начиная с самого высокого элемента. Когда уровень заполнен, следующий элемент укладывается на первый элемент текущего уровня и начинается укладка нового уровня. В рамках задачи трёхмерной укладки можно было бы переформулировать алгоритм следующим образом: контейнеры сортируются по невозрастанию высоты, затем производится их последовательная укладка начиная от начала координат, когда сумма ширины каждого блока текущего уровня становится меньше или равна ширине контейнера производится переход на новый уровень, то есть для первого слоя координата Х1 = 0, для второго максимальной глубине среди элементов первого X2 = max(X1;i). Направление осей координат приведено на рис. 1.

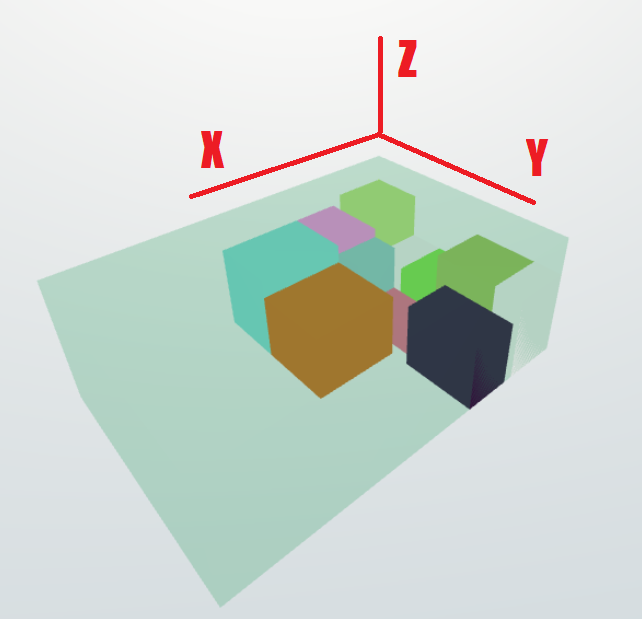


Рис. 1. Контейнер укладки и оси координат

Алгоритм NFDH малоэффективен из-за полостей образующихся между рядами, кроме того в рамках трёхмерной задачи не учитывается свободное пространство в направлении оси Z. Исходя из этого для алгоритма существуют некоторые модификации:

* First Fit Decreasing High – (FFDH). После заполнения уровня по правилам NFDH, в массиве элементов ищется первый подходящий по ширине элемент, для заполнения оставшегося места по оси Y.
* Best Fit Decreasing High – (BFDH). В отличии от FFDH ищется наиболее подходящий элемент.
* Floor Ceiling No Rotation – (FCNR). Этот алгоритм производит укладку уровня с помощью BFDH, затем в оставшиеся полости укладывает подходящие элементы, причем укладка производится на «потолок» текущего уровня (рис. 2).

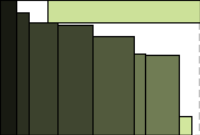


Рис. 2. Укладка с помощью FCNR

Наиболее результативным и лучше укладывающим среди рассмотренных алгоритмов будет FCNR **[]**. В следующей главе будет представлена модификация FCNR для рассматриваемой задачи.

* 1. ***Решение на основе уровневого алгоритма***

Описанный алгоритм FCNR можно улучшить несколькими способами. Опишем подробнее его интерпретацию в трёхмерном случае. В начале работы алгоритма имеем

(1.1)

(1.2)

(1.3)

где – набор упаковок которые описываются метриками , , – соответственно ширина, высота и глубина. – координата х текущего слоя, а - координата очередной упаковки слоя.  *-* метрики контейнера. В углу контейнера, в который будут укладываться расположим оси координат как на рис. 1.

Упорядочим по убыванию высот упаковок и получим – отсортированное множество. Далее последовательно осуществляем укладку уровней. В укладке уровня в данном случае можно отметить три действия. Первое и второе смежны, а третье выполняется после всех.

Суть первого действия: берётся первая упаковка из (то есть с максимальной высотой) и устанавливается в контейнер углом в точку с минимально допустимыми (не занятой другим контейнером) координатами .

(1.4)

тут – очередная упаковка, которая включает в себя метрики в и координаты , которые соответствуют координатам х, y, z внутри контейнера. Если же то слой завершен и следует перейти к следующему, если , то алгоритм завершен. При переходе к следующему слою

(1.5)

то есть, увеличивается на величину равную максимальной глубине среди блоков уровня, а координата обнуляется. Если уровень не завершен, то .

Суть второго действия: после установки очередного блока сверху на него устанавливается ещё один, с учетом того, что его основание должно вмещаться в основание нижестоящего блока. После установки переходим к первому действию.

Суть третьего действия: после построения очередного уровня и обновления , заметим, что из-за разных глубин у уложенных коробок образуются полости между линией разграничения уровня, определяющейся , и уложенным блоком (если это не блок с максимальной глубиной на уровне). Для их заполнения последовательно перебираются блоки уровня, и на каждом из них определяется

(1.6)

где - разница между максимальной глубиной и глубиной текущего блока. Далее в ищется первый элемент, который удовлетворяет следующим условиям: его ширина меньше либо равна ширине текущего блока, а глубина меньше либо равна . Данная «доупаковка» производится для каждого элемента уровня. После укладки каждой коробки она удаляется из .

Во время выполнения действий производится подсчёт реального объёма упаковки .

(1.7)

(1.8)

(1.9)

(1.10)

Объём разбит на составляющие по действиям. Объём соответствует объёму блоков уложенных в первом действии, но с общей глубиной , которая максимальна на уровне, – ширина и высота очередного блока. Объём – блока уложенного сверху, но за площадь основания берётся основание нижележащего блока - , высота блока . Из суммы предыдущих нужно вычесть сумму объём блоков, установленных на третьем этапе - . Таким образом учитываются пустоты между уровнями и их компенсация на третьем этапе. На рис. 3. можно увидеть упаковку данным алгоритмом.

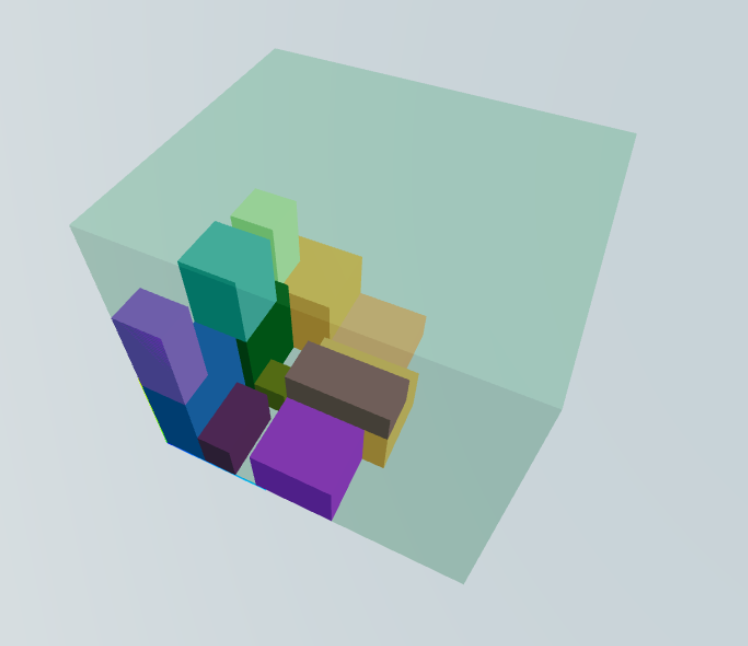


Рис. 3. Упаковка разработанным уровневым алгоритмом

* 1. ***Генетический алгоритм***

Кроме рассмотренных уровневых алгоритмов в задаче укладки часто используется достаточно известный и популярный эвристический подход – генетический алгоритм. Данный подход позволяет корректировать время, затраченное на поиск, и допустимый порог точности решения задачи, выбирая приемлемую величину по времени и точности. Суть генетического алгоритма заключается в следующем:

1. В начале работы алгоритма формируются хромосомы. Хромосомы это – наборы данных, которые представляют собой решения задачи. Чаще всего содержимое стартовых хромосом формируется случайным образом. Также отдельным пунктом стоит задача преобразования решения задачи в такую форму, которая была бы удобна для мутаций и скрещиваний.
2. Далее производится естественный отбор на протяжении N поколений, где N выбирается в зависимости требований по времени и точности решения. На каждой итерации отбора осуществляется скрещивание хромосом.
3. Скрещивание выполняется с помощью функции скрещивания, которая выбирается в зависимости от задачи. Результатом её применения к двум хромосомам будет являться новое решение. Применяя данную функцию к выбранным особям нужное количество раз получается новая популяция.
4. Кроме того, вводится операция мутации. С малой вероятностью она меняет часть хромосомы новой популяции.

* 1. ***Решение задачи на основе генетического алгоритма***

Решение задачи трёхмерной укладки сопряжено с определёнными трудностями в связи с потребностью преобразования условий задачи в термины генетического алгоритма. Для решения задачи нужно определить, что есть хромосома в данном случае. Поскольку нам нужно оптимизировать укладку в объём, то решением задачи была бы последовательность X из упаковок, которая после укладки определённым алгоритмом давала бы при подсчёте целевой функции по формуле (1) оптимальное значение, в нашем случае наилучшем будет

(2.1)

где K отношение суммы объёмов упаковок к объёму итоговой упаковки.

То есть, в данном случае хромосома представляет собой последовательность из упаковок

(2.2)

где X – хромосома, а – упаковка или же контейнер, которая определяется по следующей формуле:

(2.3)

width, height, depth – соответственно ширина, высота, глубина упаковки .

На этапе генерации стартовых хромосом должен получится массив хромосом (поколение) с хаотичным порядком упаковок в каждой хромосоме

(2.4)

G – поколение, а Xi хромосома со своим порядком укладки

(2.5)

Резюмируя вышеописанное, результатом работы первого этапа алгоритма должен быть кортеж G. Для дальнейшего вычисления целевой функции по формуле (1) требуется сумма объёмов упаковок, которую так же можно подсчитать на первом этапе.

(2.6)

где Vsum – сумма объёмов упаковок, которая используется на каждой итерации отбора.

На втором этапе генетического алгоритма должен производится естественный отбор N поколений. Опишем подготовку данных перед скрещиванием. В начале итерации поколения имеем – хромосомы поколения полученного на предыдущем этапе отбора или же сгенерированное на первом этапе алгоритма. Далее для оценки приспособленности требуется вычислить целевую функцию для каждой хромосомы, для этого осуществляется укладка (рис. 7. блок-схема алгоритма укладки) по описанному в хромосоме (i – индекс поколения, j – хромосомы в множестве поколения) порядку и в процессе укладки подсчитывается (далее *volume*– объём в который была произведена упаковка (с учётом пустот) и используя его производится вычисление целевой функции. Получаем популяцию с текущими хромосомами .

(2.7)

Pi – текущее поколение, – представитель популяции.

(2.8)

где – хромосома из которой был составлен представитель популяции, – содержит координаты укладки упаковок в том же порядке, в котором они указаны в , – вычисленное значение целевой функции. Далее, упорядочим популяцию в порядке невозрастания по и получим упорядоченную популяцию .

После того как был получен набор можно приступить к отбору. В данной работе предлагается отбросить две самые неудачные особи в случае небольшой выборке данных (около 30 упаковок) и 7% при большем количестве входных данных. Аналогичное количество (2 или 7%) особей переходит в следующее поколение. В множество хромосом нового поколения войдут хромосомы лучших особей из отбора и особей, полученных в результате скрещивания над множеством хромосом из которого исключены хромосомы особей, непрошедших отбор.

Выбор пары особей для скрещивания проводится турнирным способом. Турнирный способ подразумевает выбор нескольких особей (более двух) и сравнение значений целевой функции у каждой из них – для двух лучших осуществляется скрещивание. В текущей реализации алгоритма выбирается три особи. Выбор особей может быть детерминированным (например, среди каждых трёх по порядку определяемом порядком кортежа популяции) или же случайным. В нашем случае остановимся на случайном, т.к. он даёт большую вероятность того, что важные фрагменты хромосом не будут утеряны.

Скрещивание производится следующим образом:

* Случайным образом выбирается точка скрещивания. Точкой является число в диапазоне размера хромосомы, обозначим его *I*.
* В результирующую хромосому попадает часть хромосомы первого родителя до индекса равного *I* включительно, остальная часть заполняется из хромосомы второго родителя.
* Для заполнения второй части результирующей хромосомы требуется из хромосомы второго родителя исключить упаковки, которые находятся в первой части результирующей хромосомы, а затем оставшиеся упаковки следует последовательно дописать в результат.

После завершения процедуры скрещивания проводится операция мутации над каждой хромосомой нового поколения с вероятностью 1%. В контексте текущей задачи мутация проходит в виде перестановки местами первого и случайно выбранного элемента хромосомы. Лучшие хромосомы отбора защищены от мутаций.

(2.9)

где – новое поколение хромосом полученное путём итерации генетического алгоритма.

После получения и соответственно алгоритм считается выполненным. Результатом будет – первая особь популяции, в которую согласно (2.8) включено значение целевой функции, которое будет наилучшим за все поколений. Далее описана блок схема построенного алгоритма на рис. 4. Мутации и скрещивания были для удобства выделены в отдельные схемы. На рис. 5. и рис. 7. соответственно видим блок-схему скрещивания и мутации. Для хаотичности порядка укладки у стартовых хромосом используется алгоритм тасования Фишера – Йетса, который генерирует случайную перестановку элементов массива (рис. 6).

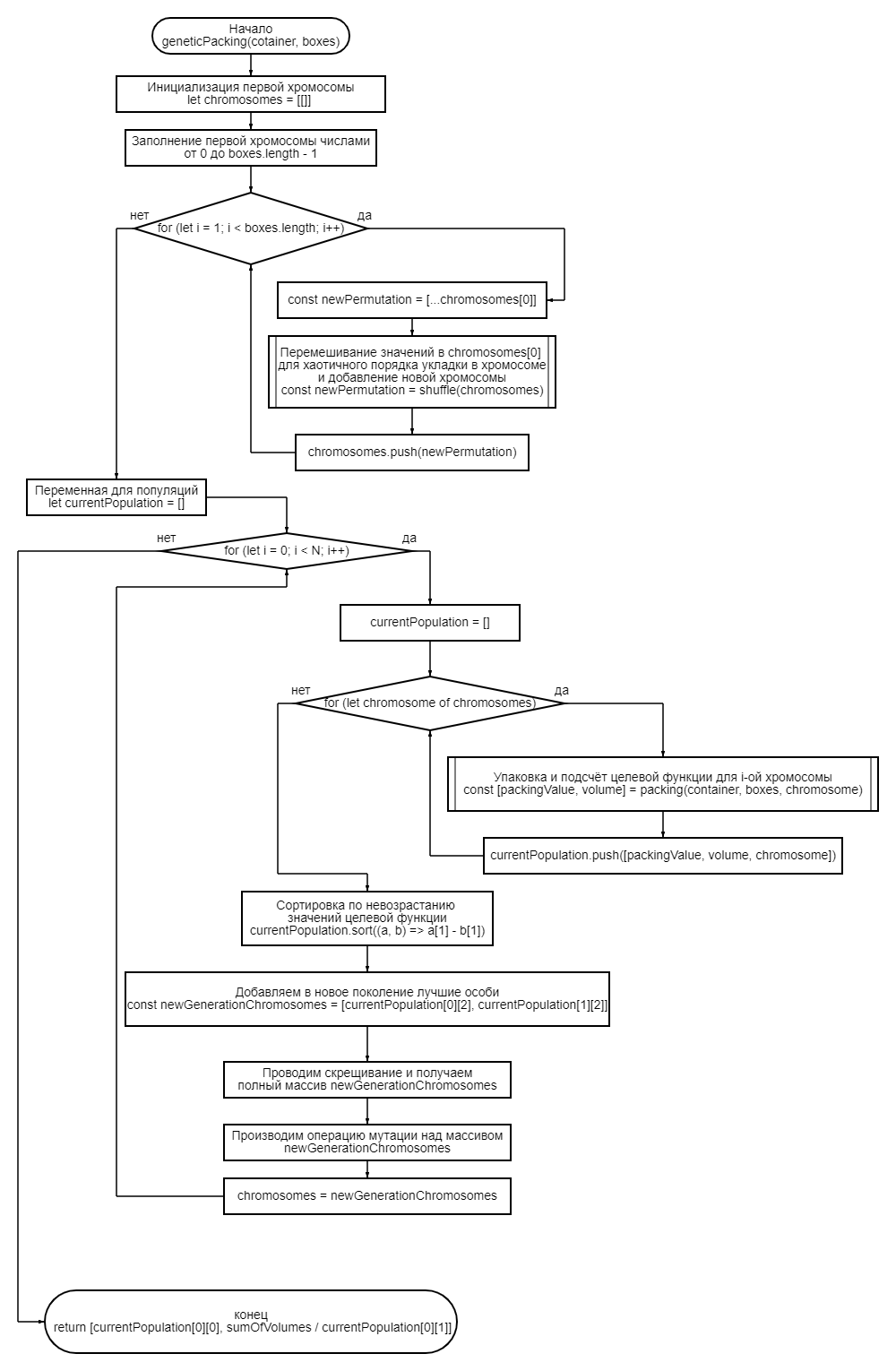
**

Рис. 4. Блок-схема генетического алгоритма укладки

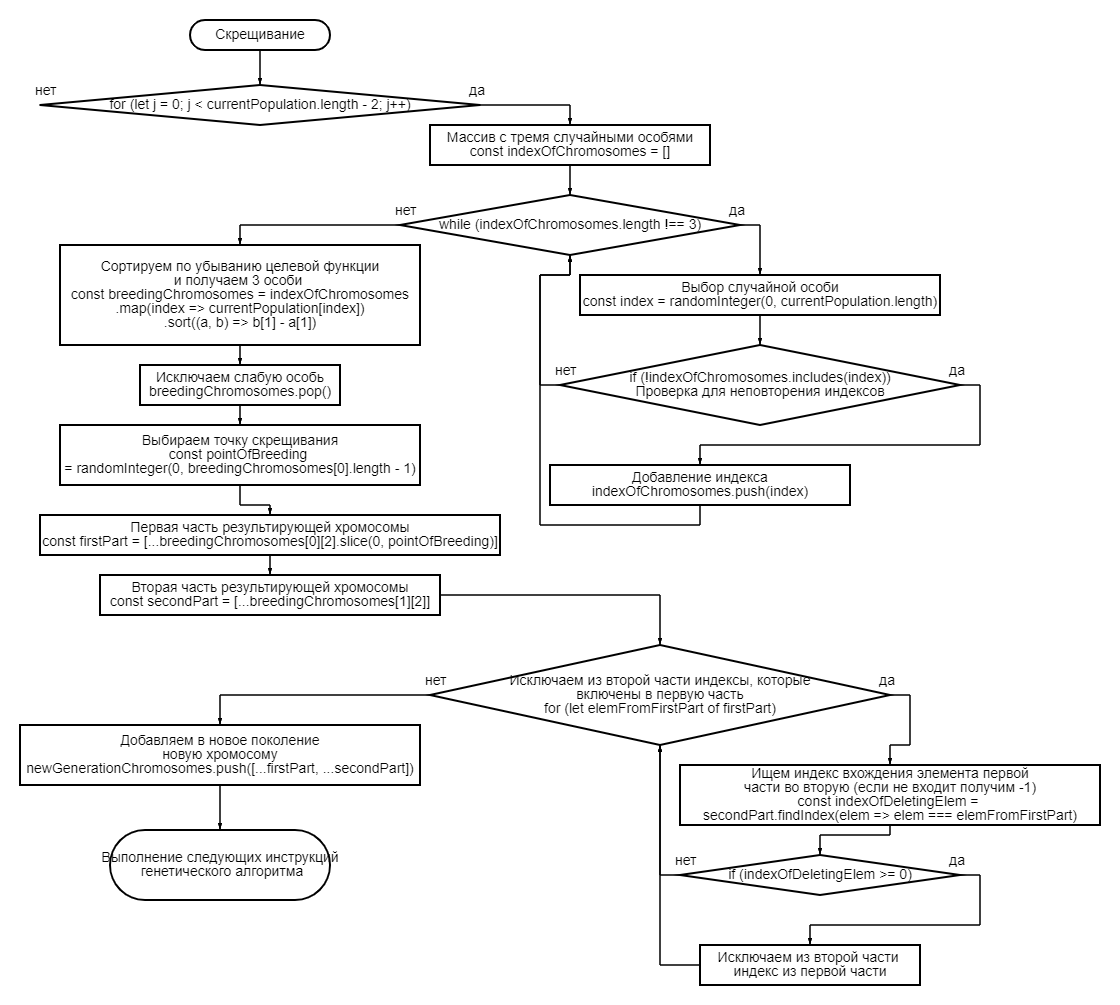


Рис. 5. Блок-схема скрещивания (часть блок-схемы на рис. 4)

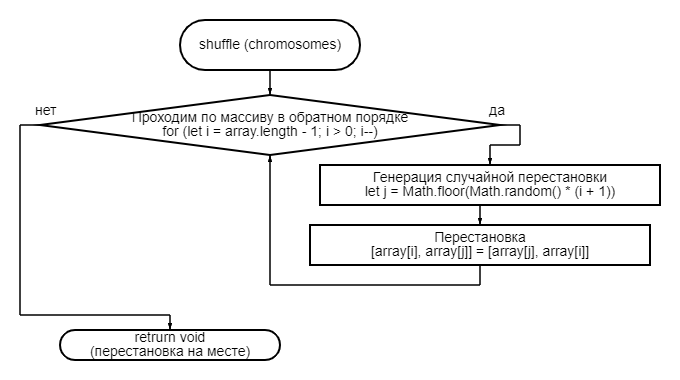


Рис. 6. Блок-схема алгоритма перестановок

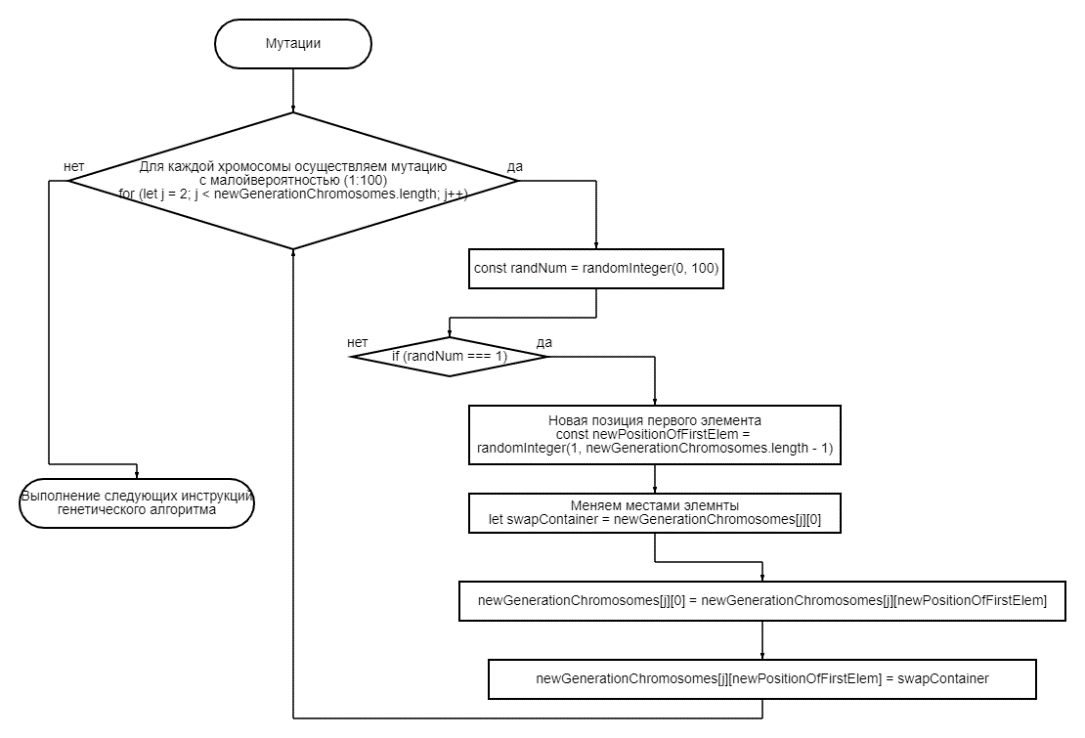


Рис. 7. Блок-схема мутаций (часть блок-схемы на рис. 3)

Кроме частей алгоритма, которые указаны выше осталась одна не описанная и важная деталь – алгоритм укладки. Алгоритм является уровневым, но со следующей модификацией: после добавления в уровень очередного блока на него проводится укладка подходящих упаковок сверху. Критерий установки сверху – основание вышестоящего блока вписывается в основание нижестоящего, что значит, что ширина и глубина вышестоящего меньше либо равна ширине и глубине нижестоящего. Рост уровня вверх ограничен высотой контейнера, а рост вправо шириной контейнера. Сортировки как в алгоритме из раздела 2.1.2 не производится, как и «доупаковки», поскольку оптимизационную роль играет генетический алгоритм. Полное описание укладки (блок-схема) представлено на рис. 9. Пример укладки генетическим алгоритмом на рис. 8.

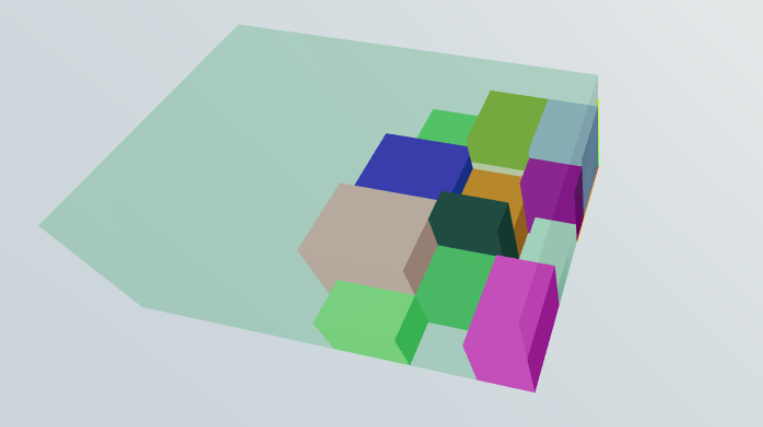


Рис. 8. Укладка генетическим алгоритмом

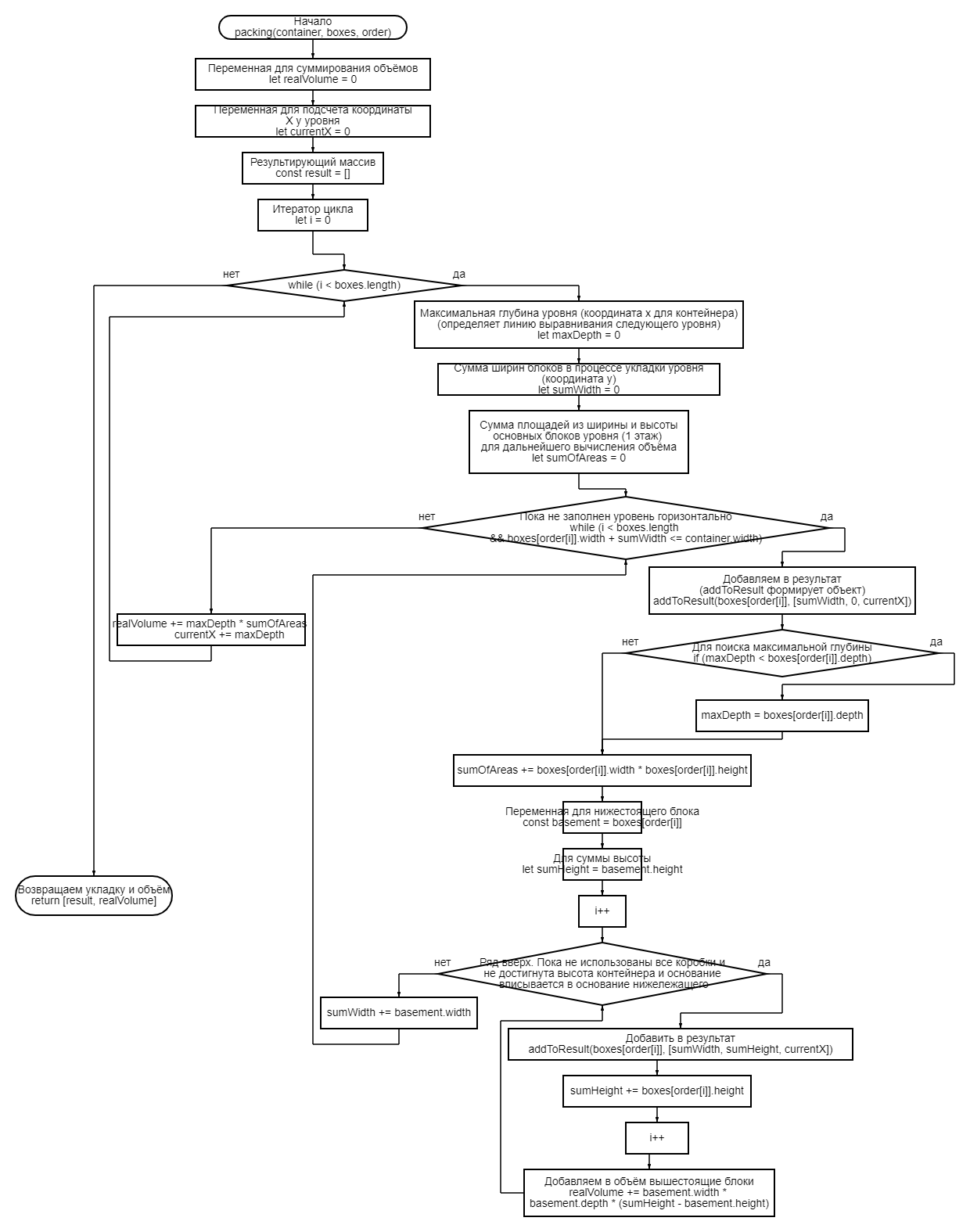


Рис. 9. Блок-схема алгоритма укладки

1. **Web-приложение «Укладка»**

Данное приложение было разработано с целью автоматизации и систематизации укладки на рис. 10 можно увидеть основные случаи использования разрабатываемого продукта.

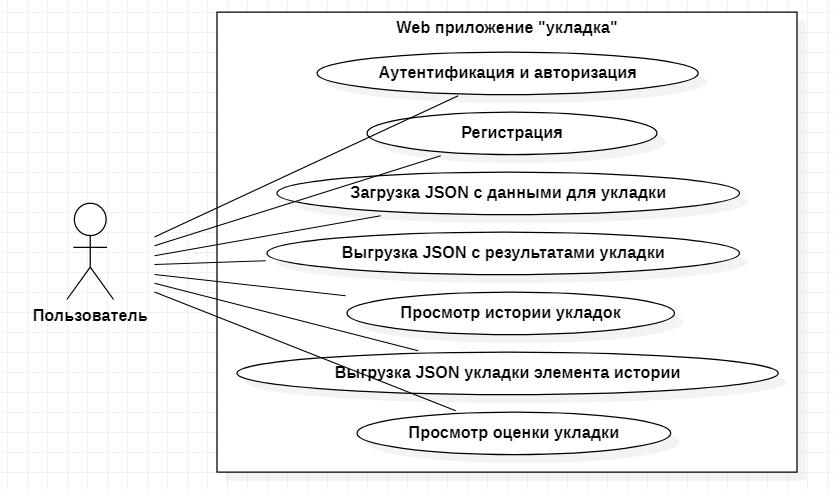


Рис. 10. Use case диаграмма web-приложения «Укладка»

* 1. ***Элементы приложения***

Исходя из поставленных задач и use case диаграммы следует выделить части приложения и уточнить детали их реализации.

* Для реализации сохранения результатов работы алгоритма требуется место для хранения данных, то есть база данных. Кроме того, для разграничения массивов данных, получившихся по итогам укладок, и сохранения конфиденциальности результатов требуется введение пользовательской модели приложения. Рассмотрим это в 1.1.1.
* Для реализации укладки требуется написать алгоритм на языке JS, кроме того, для удобства пользования следует разработать визуализацию укладки и таблицу с деталями укладки.
* Для удобства демонстрации работы укладки, сбора статистических данных требуется реализовать случайную генерацию входных данных для укладки.
* Для экспорта и импорта данных в формате JSON требуется реализовать парсинг JSON в объекты укладки и обратно, а также предоставить интерфейс пользованиями этими функциями.
  1. ***Firebase и бизнес-логика.***

Как было сказано во введении, из-за небольших требований к функциональности базы данных и из-за большой скорости разработки, для реализации авторизации с аутентификацией и хранением данных работы алгоритма была выбрана Firebase. Поскольку Firebase нереляционная база данных, то в ней вместо таблиц хранятся коллекции документов в формате JSON, кроме того, объекты в JSON могут быть вложенными. На рис. 11 описана схема базы данных приложения «Укладка». Существует две коллекции – пользователи и укладки. На схеме приведены названия только внешних объектов, полные схемы объектов приведены на рис. 12.

* Пользователи (users):
  + email – адрес электронной почты пользователя;
  + password – пароль;
  + uid – уникальный идентификатор пользователя и первичный ключ.
* Укладки (palletings):
  + timestamp – время в миллисекундах с 01.01.1970 на момент создания укладки. Является первичным ключом;
  + userId – внешний ключ к коллекции пользователей;
  + palleting – JSON содержащий объект с описанием укладки;
  + density – процент оптимальности укладки;
  + container – JSON описание контейнера, в которой осуществилась укладка.

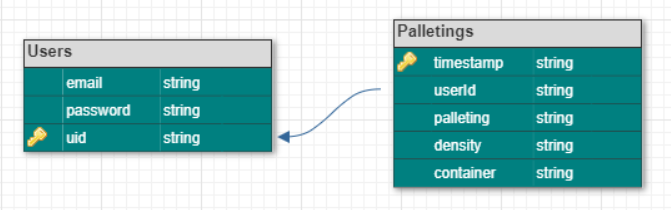


Рис. 11. Схема базы данных

Поскольку React.js придерживается ФП, то далее приводятся схемы основных объектов бизнес-логики приложения, в смысле структур данных или моделей данных, а не классов в понимании ООП (рис. 12).

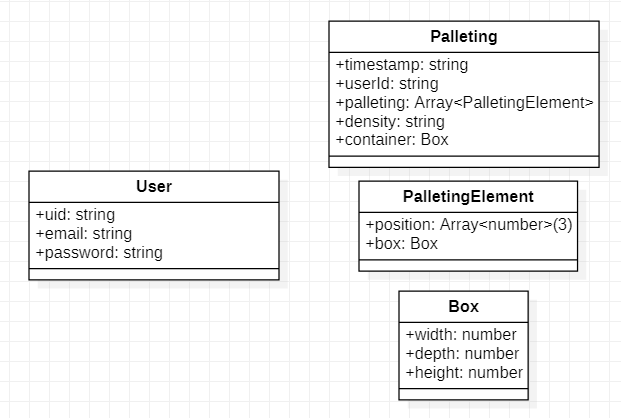


Рис. 12. Схема моделей данных

На рис. 12 можно увидеть схему Palleting, которая описывает документ коллекции Palletings (рис. 11). В нём palleting – укладка описывается массивом элементов укладки – Array<PalletingElement>. В свою очередь элемент укладки PalletingElement представляет собой:

* position – позиция в контейнере укладки состоит из массива трёх координат x, y, z.
* box – элемент укладки, упаковка которая описывается объектом Box:
  + width – ширина;
  + depth – глубина;
  + height – высота;
    1. ***Авторизация и аутентификация***

Для взаимодействия с Firebase из React предоставляется специальное API посредством пакета “firebase”. Перед инициализацией приложения требуется инициализировать объект firebase, с помощью которого можно производить действия над базой данных и также инициализировать объекты firebaseAuth (авторизация) и firebaseFirestore (хранилище данных). С помощью React.Provider можно осуществить инъекцию глобальных переменных в приложение, что как раз и требуется в данном случае.

С помощью объекта firebaseAuth и хука onAuthStateChanged, который позволяет отслеживать изменение текущего пользователя, можно динамически менять интерфейс. Например, с его использованием осуществляется приватность доступа к страницам приложения, не авторизованный пользователь не сможет пользоваться приложением. Также предоставляются следующие функции:

* signInWithEmailAndPassword – позволяет аутентифицироваться пользователю в системе по почте и паролю.
* createUserWithEmailAndPassword – создает пользователя с почтой и паролем.
* signOut – осуществляет выход из аккаунта.

Пользуясь вышеописанными функциями производится реализация работы в приложении от имени созданного аккаунта (рис. 13).

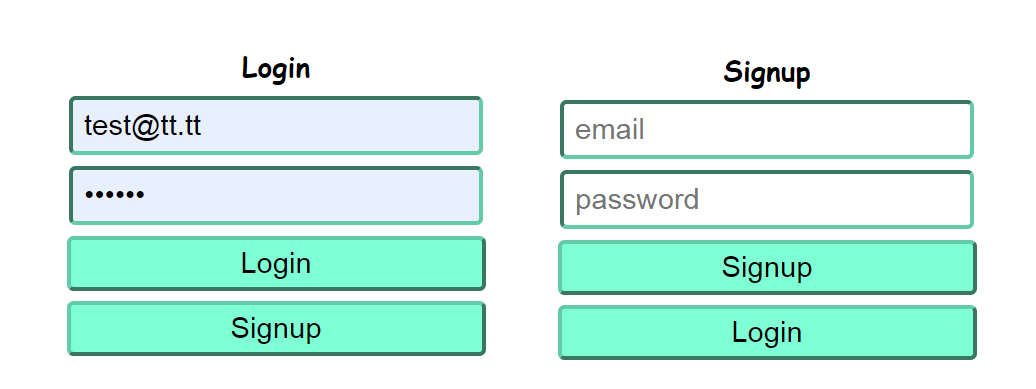


Рис. 13. Формы входа в приложение и создания нового аккаунта

* + 1. ***Журнал учёта укладок***

После аутентификации пользователь видит главную страницу приложения (рис.14.). Далее, он может кликнуть по кнопке Menu и увидеть меню приложения (рис 15). В нём пользователь может просмотреть статистику своих укладок и по двойному клику скачать JSON с описанием укладки.

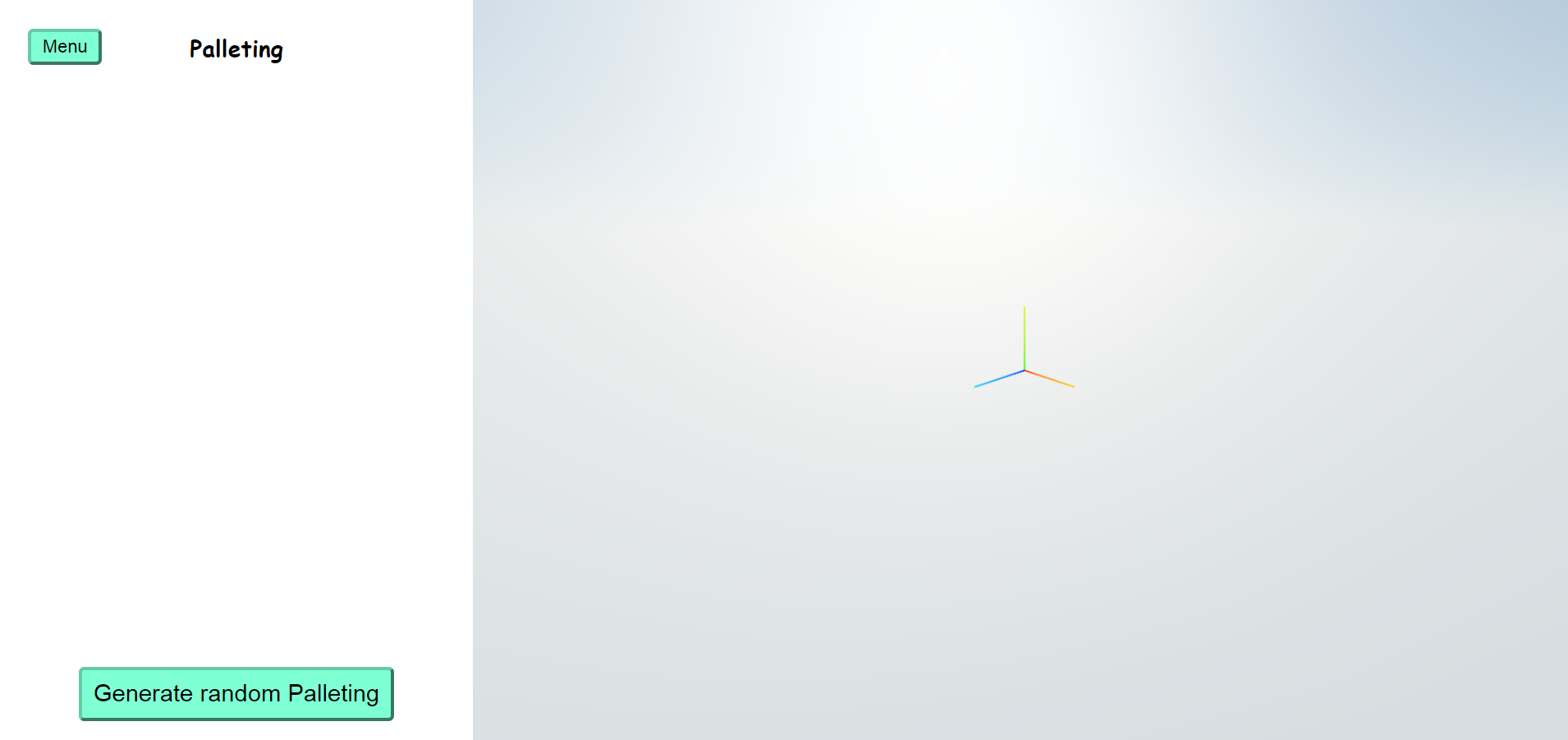


Рис.14. Главная страница приложения



Рис. 15. Меню приложения.

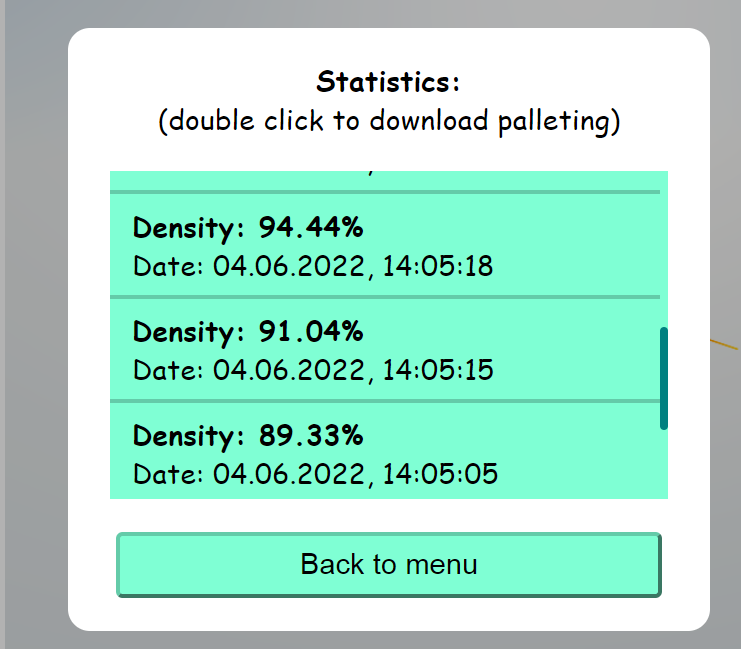


Рис. 16. Список с историей укладок

Статистический список представленный на рис. 16 поддерживает пагинацию, а точнее скроллирование с подгрузкой новых порций данных. Список упорядочен по убыванию времени. Со стороны взаимодействия с firebase процесс выглядит следующим образом.

* В первую очередь, с помощью firebaseFirestore и функции collection создаётся ссылка на коллекцию palletings.
* Затем, при каждом срабатывании алгоритма укладки с помощью функции firestore addDoc(collection, id) добавляется новая укладка в базу данных.
* При открытии статистики загружаются первые элементы используя функцию firestore query(collection, where, orderBy, limit), где where – предикат аналогичный ключевому слову в языке sql, orderBy – порядок сортировки и поле по которому производится сортировка, limit – количество которое требуется загрузить. Далее, используется функция onSnapshot(query, callback), которая использует запрос query и передает результат аргументов в callback. По завершении загрузки, в коллбеке устанавливается новое значение для массива, который отображается в списке и запоминается последний загруженный элемент.
* После достижения скролла нижней границы блока производится подгрузка новой порции данных таким же способом, как в прошлом пункте, только добавляется дополнительный предикат в запрос – startAfter(lastElem), который указывает начиная с какого элемента подгружать данные.
  1. ***3D – визуализация укладки***