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 задачи трёхмерной укладки

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»……………..…..……………….………. | 22 |
| 2.1. Элементы выбранных технологий………………………………. | 24 |
| 2.2.1. Функциональные компоненты React.………………………. | 24 |
| 2.2.2. Роутинг в React.……………………………………………… | 25 |
| 2.2.3. Three.js.……………………………………………………….. | 26 |
| 2.2. Элементы приложения…………………………………………… | 26 |
| 2.3. Firebase и бизнес логика…………………………………………. | 28 |
| 2.3.1. Авторизация и аутентификация……………………………. | 30 |
| 2.3.2. Журнал учёта укладок………………………………………. | 31 |
| 2.4. 3D – визуализация укладки……………………………………… | 33 |
| 2.5. Реализация алгоритмов в приложении………..………………… | 34 |
| Заключение……………………..………………………………………….. | 36 |
| Список использованных источников………… | 38 |
| Приложение………… |  |

**Список сокращений**

**JS** – JavaScript.

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

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

**URL** - Uniform Resource Locator – уникальный указатель ресурса.

**JSP** – Java server pages.

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

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

**UI** – user interface – интерфейс приложения, предоставляющийся пользователю.

**JSX** – расширение синтаксиса JS, позволяющее подставлять в html шаблон выражения языка JS.

**RGB** – цветовая модель для кодирования цвета с помощью трёх цветов (red – красный, green – зелёный, blue – синий).

**SPA** – single page application.

**CSS -** cascade style sheets – каскадные таблицы стилей.

**Sass –** super awesome style sheets – препроцессор для CSS.

**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. Также актуальна задача паллетизации на складах [15], которая в целом схожа с укладкой в кузов и различается только в параметрах, которые требуется опитимизировать.

Вышеописанные задачи могут быть сведены к задаче трёхмерной укладки коробок [8] (параллелепипедов) в заданный кузов (объём). Данная задача давно известна и глубоко исследована в математике и является NP полной [10], [6] 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 [1]. React позволяет динамически отрисовывать элементы страницы, в зависимости от данных бизнес логики (хуки useState, useEffect). Эта функция будет полезна 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 близкой к оптимальной трёхмерной укладки грузов, и приложения, которое предоставляет интерфейс для введения данных укладки и просмотра результата работы алгоритма.

При выполнении работы необходимо решить следующие задачи:

* В процессе работы над реализацией 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 используя подсчёт объёмов;
  + Оценить оптимальность на выборке случайно сгенерированных данных;
  + Создать генетический алгоритма трёхмерной укладки.
* В процессе разработки web-приложения:
  + Создать интерфейс для загрузки данных о контейнере и упаковках;
  + Реализовать экспорт данных результата работы алгоритма укладки в формате JSON;
  + Реализовать генератор случайных входных данных для алгоритма укладки. То есть, должна происходить генерация некоторого количества контейнеров с их размерами (случайные числа должны быть ограничены диапазоном);
  + Создать 3D просмотр результата работы алгоритма;
  + Реализовать авторизацию и создание аккаунта для пользователя;
  + Реализовать сохранение результатов работы алгоритма для каждого пользователя в базе данных.
* Кроме того требуется:
  + Освоить базовую функциональность Firebase;
  + Освоить базовую функциональность Three.js.

1. **Алгоритм укладки**

Во введении было обосновано, что в решении задачи трёхмерной укладки в прикладном смысле наиболее применимым будет решение в виде эвристического алгоритма. 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 [3], укладывая их либо друг на друга или же последовательно, пока не будет заполнена плоскость основания, далее выстраивается новый слой, для которого основанием будет предыдущий. Слои могут выстраиваться на нижнем основании контейнера или же боковом. 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:

(1.1)

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 [9], [12], 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.

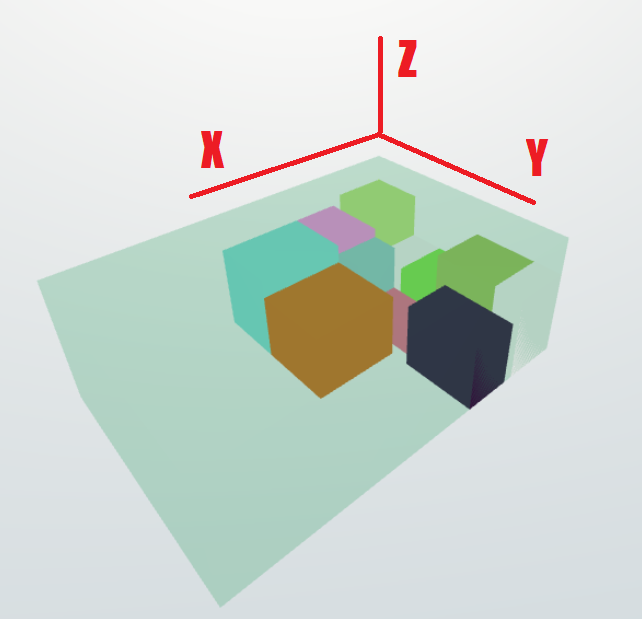


Рис. 1. Контейнер укладки и оси координат

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).

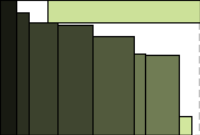


Рис. 2. Укладка с помощью FCNR

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

(1.2)

(1.3)

(1.4)

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 .

(1.5)

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

(1.6)

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

(1.7)

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 .

(1.8)

(1.9)

(1.10)

(1.11)

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.

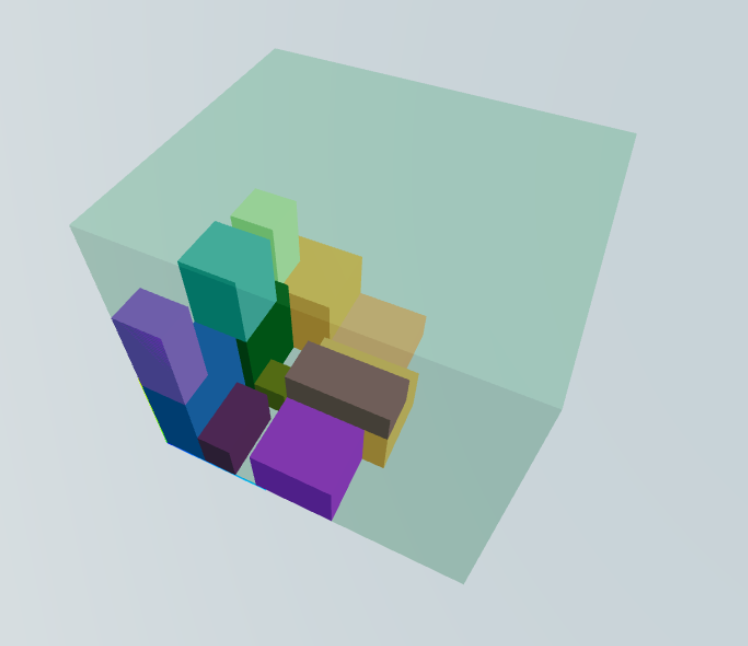


Рис. 3. Упаковка разработанным уровневым алгоритмом

* 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 [14]. Стандартные подоходы к решению задачи укладки генетическим алгоритмом приведены в [4]. 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 (1.1) 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).

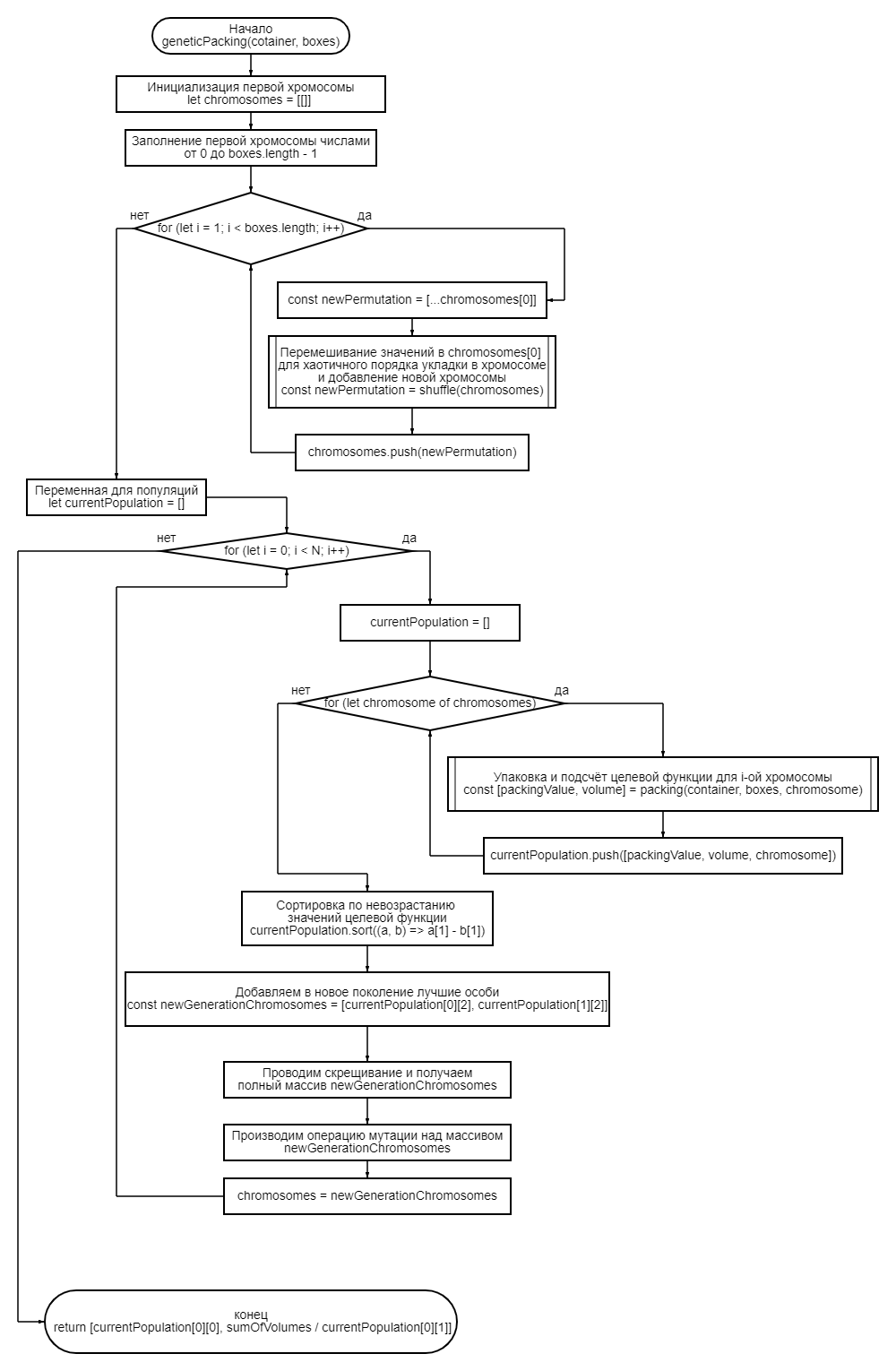
**

Рис. 4. Блок-схема генетического алгоритма укладки

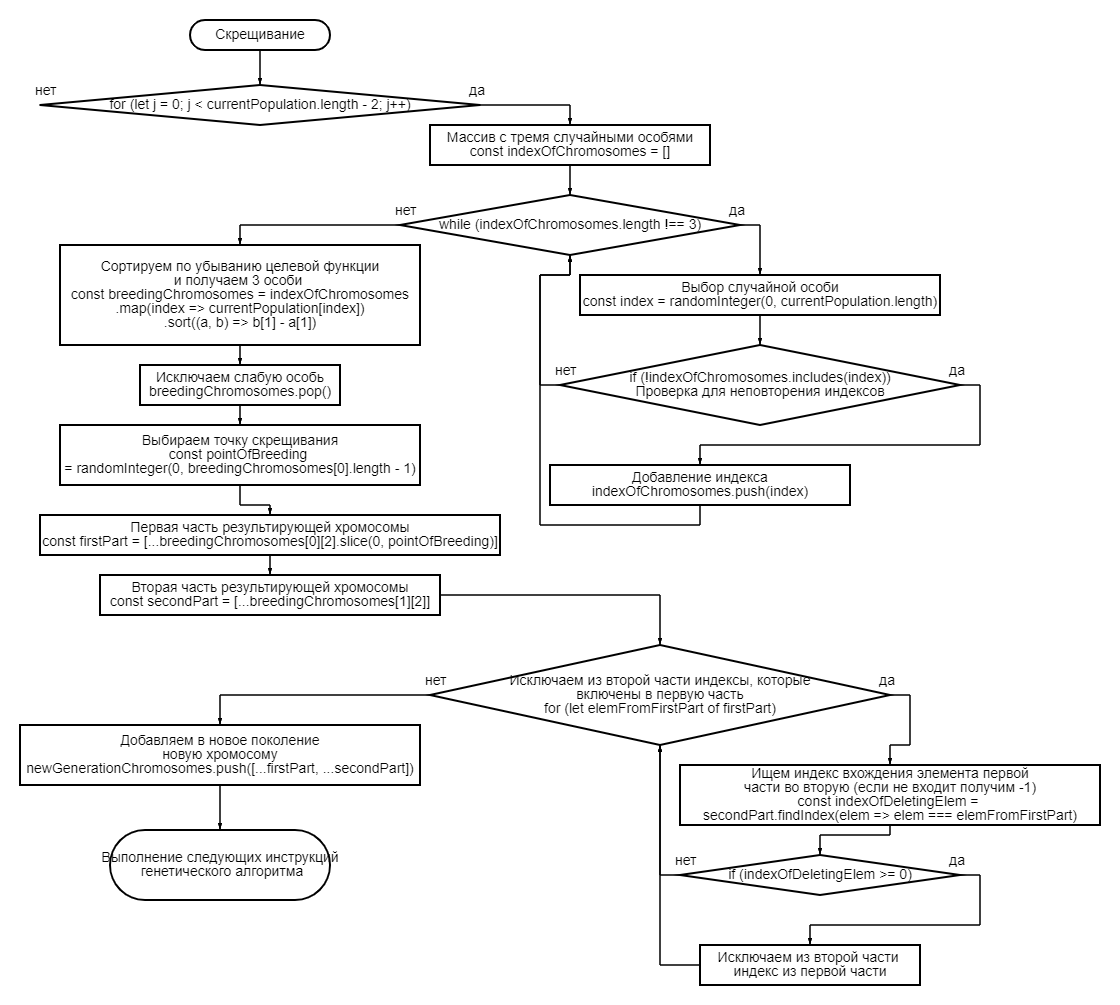


Рис. 5. Блок-схема скрещивания (часть блок-схемы на рис. 4)

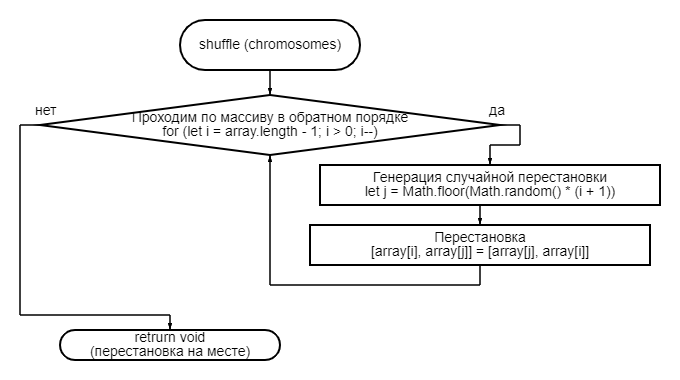


Рис. 6. Блок-схема алгоритма перестановок

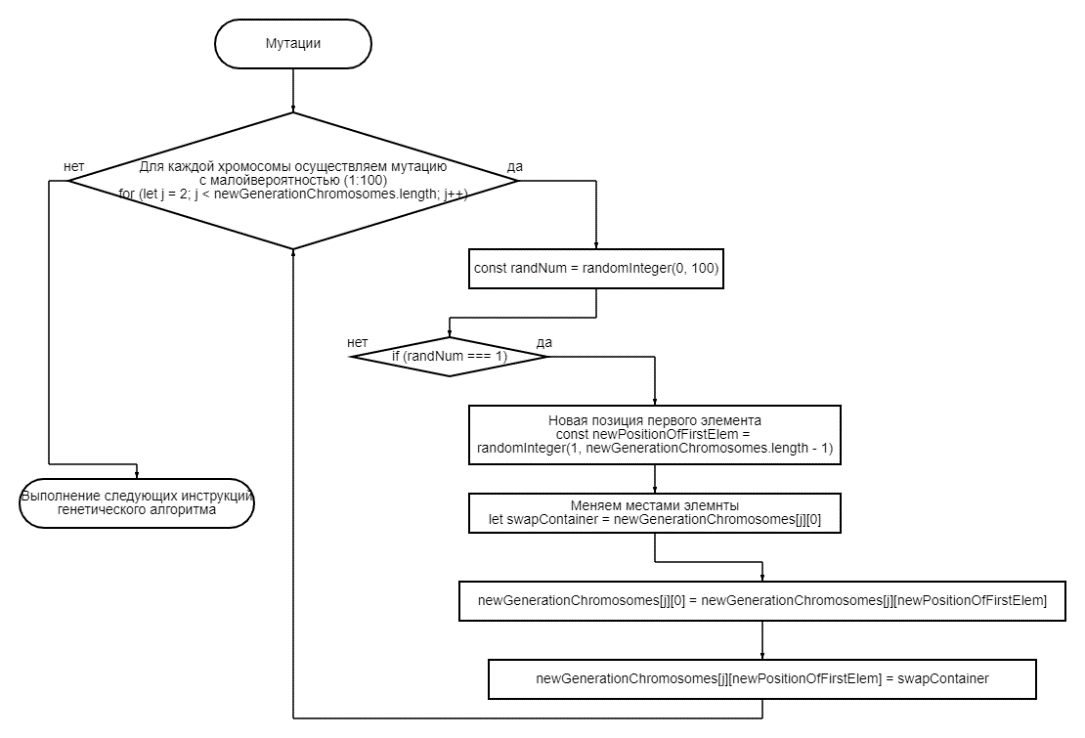


Рис. 7. Блок-схема мутаций (часть блок-схемы на рис. 3)

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.

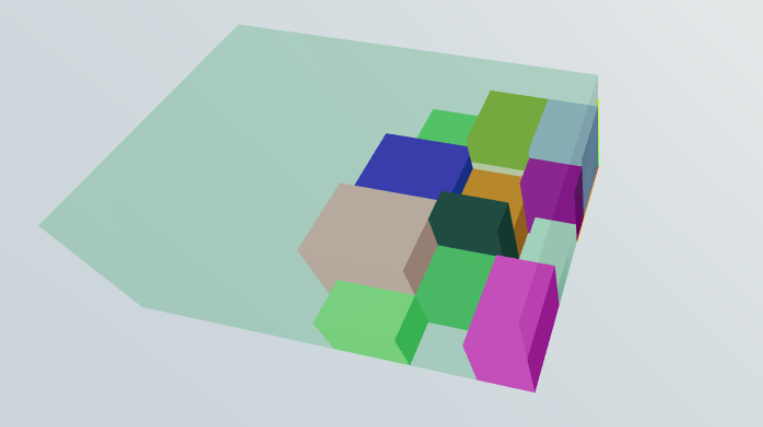


Рис. 8. Укладка генетическим алгоритмом

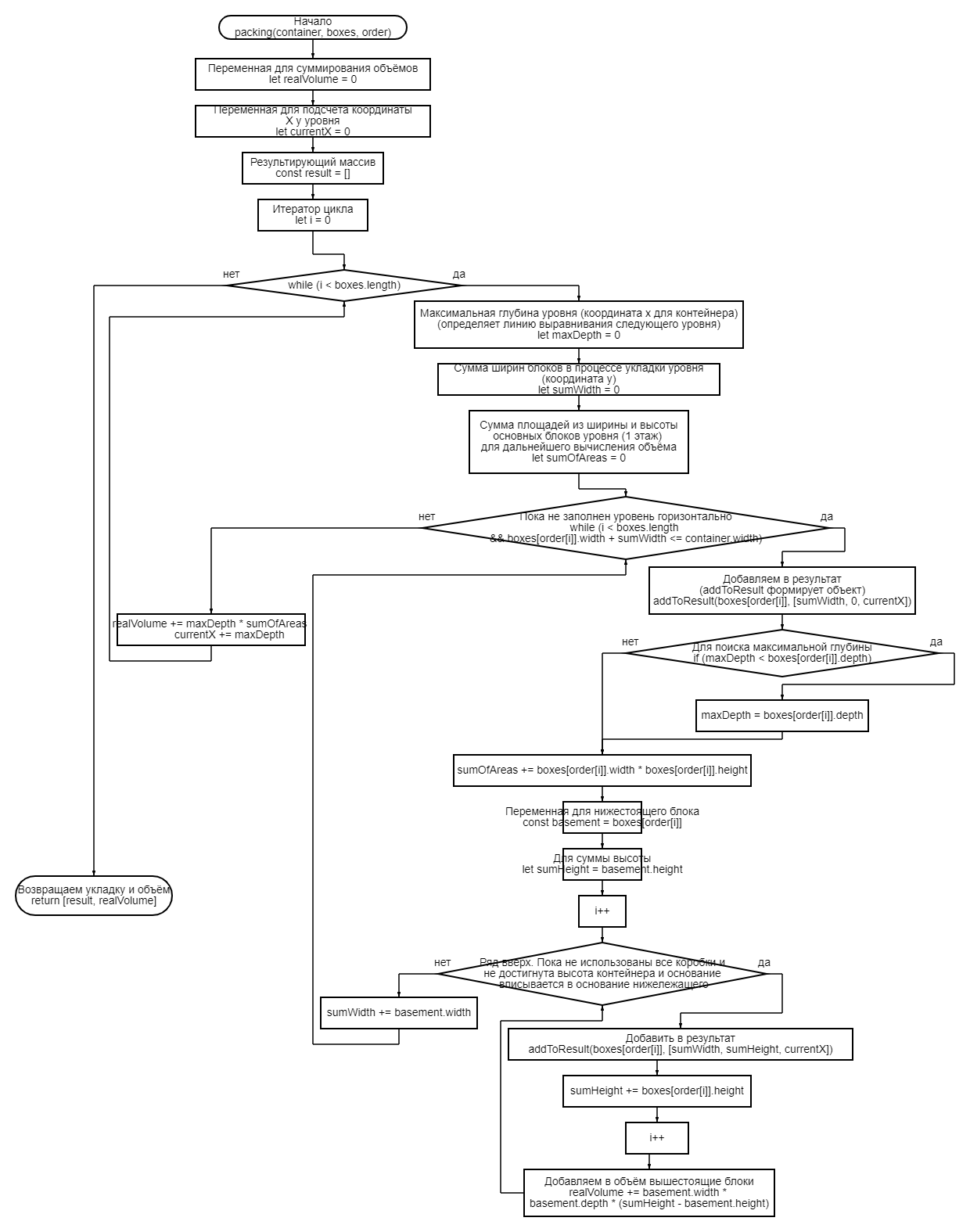


Рис. 9. Блок-схема алгоритма укладки

1. **Web-приложение «Укладка»**

Web-приложение «Укладка» было разработано с целью автоматизации и систематизации укладки, а также создания её графического представления. На рис. 10 можно наблюдать основные случаи использования разрабатываемого программного обеспечения.

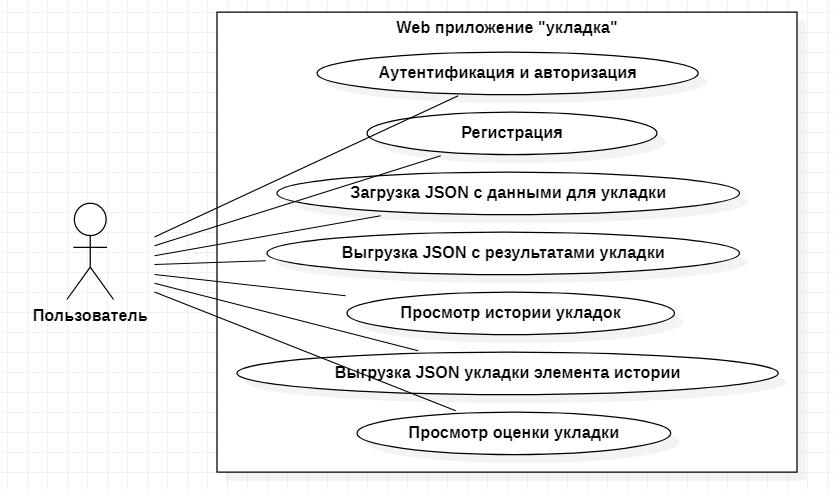


Рис. 10. Use case диаграмма web-приложения «Укладка»

Архитектура приложения представлена на UML [13] диаграмме пакетов (рис. 11). Она подразделяется на два больших раздела – React (клиентская часть и Firebase – база данных). Подробнее о каждом:

* Пакет React:
  + Ui layer – уровень пользовательского интерфейса:
    - Pages and Routes – страницы созданные с помощью шаблонов React и их роутинг (расположение по соответствующему URL) с учётом ограничений, накладываемых авторизацией;
    - CSS and SCSS [2] – стилизация страниц с помощью CSS и предпроцессора Sass. (Sass имеет два синтаксиса Sass и Scss, в работе используется второе);
    - Three.js – графическое представление укладки.
  + Business logic layer – уровень бизнес логики, другими словами – обработки данных:
    - Firestore service и Fireauth service – классы библиотеки для взаимодействия с Firebase, которые обеспечивают возможность авторизации и работы с базой данных;
    - Another useState data – данные хранимые и обрабатываемые клиентским приложением, а конкретно хуком React – useState.
  + Algorithms layer – уровень функций реализующих главную функциональность приложения:
    - Genetic packing – пакет с функцией укладки;
    - Randomize boxes – пакет реализующий генерацию случайных входных данных алгоритма.
* Пакет Firebase – логика формирования базы данных и её коллекций (рассмотренно в 2.2).

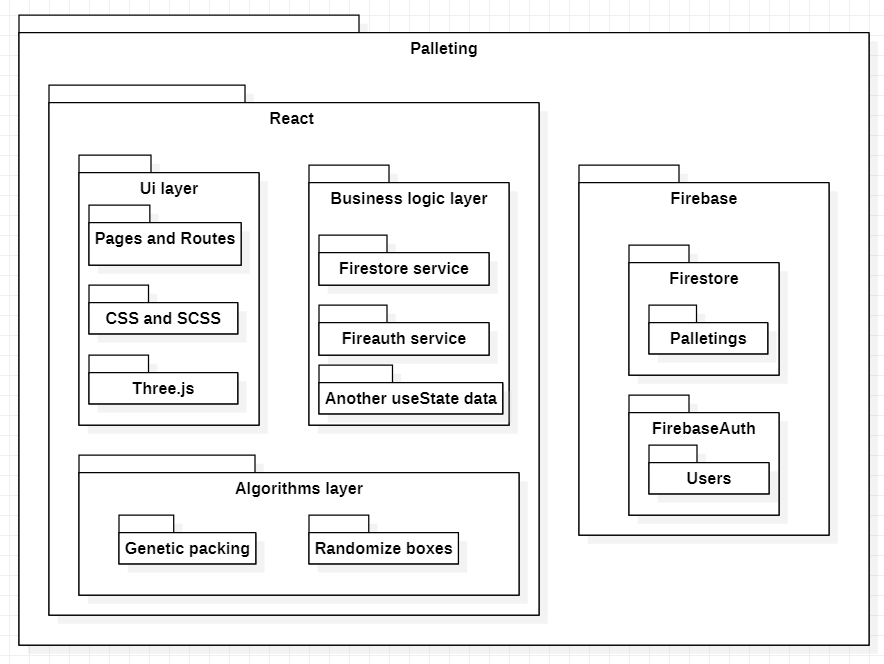


Рис. 11. Диаграмма пакетов web-приложения «Укладка»

Поскольку архитектура React-приложения выстраивается на функциях-компонентах, которые сочетают в себе UI-шаблон в виде JSX и функции для работы с бизнес логикой, то далее на рис. 12 будет приведено дерево приложения.

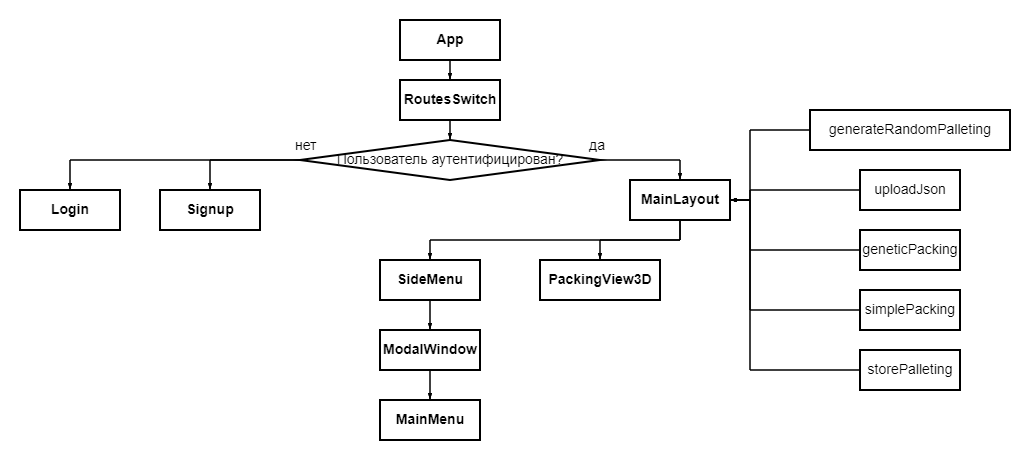


Рис. 12. Дерево компонентов приложения

* 1. ***Элементы выбранных технологий***

Перед описанием реализации приложения следует подробнее описать инструменты библиотек, выбранных для создания приложения и непосредственно использующиеся в данном web-приложении.

* + 1. ***Функциональные компоненты React***

Ранее в React до версии 16 использовались классовые компоненты, но далее было решено перейти к новому API функциональных компонентов, поскольку классовый вариант затрачивал большее количество ресурсов и памяти из-за механизмов работы классов в JS и реализации классовых компонентов в React, которая зачастую приводила к большой вложенности компонентов.

Основой архитектуры React являются компоненты, которые в данном случае являются функциями, которые в качестве аргумента получают объект props и возвращают JSX. Объект props является неизменяемым изнутри дочернего компонента. В разметке JSX возможно вставлять компоненты – в результате этого приложение представляет собой дерево компонентов (рис. 12).

Одним из главных механизмов функциональных компонентов являются хуки [22]. Хуки - это функции, которые позволяют взаимодействовать с внутренним состоянием библиотеки. В реализуемом приложении используются несколько хуков:

* useState – позволяет сохранять данные в переменной от рендера к рендеру, это требуется поскольку в отличии от классов функции существуют только в момент выполнения. Данный хук возвращает переменную и сеттер для неё. При присвоении переменной нового значения через сеттер происходит перерисовка компонента;
* useRef – позволяет хранить ссылку на объект. Ссылка существует всё время жизни компонента. В отличии от useState не влияет на перерисовку компонента;
* useEffect – позволяет вызывать переданный коллбек при изменении указанных переменных. В приложении firebase предоставляет аналоги этого хука.
  + 1. ***Роутинг в React***

Так как одной из идей React является SPA – отображение приложения на одной вкладке браузера, то в случае изменения URL не должна происходить перезагрузка страницы. Эту проблему решает библиотека React router, которая предоставляет следующие возможности:

* рендеринг страницы в случае совпадения с определённым URL -компонент Route;
* перенаправление на определённый URL – компонент NavLink;
* группировка нескольких адресов – компонент Routes.

В приложении компонент Routes используется для группировки адресов относительно аутентификации.

* + 1. ***Three.js***

Во введении было описано предназначение данной библиотеки и сказано про существование обертки над Three.js [20] для React. Это библиотека three-js-fiber. Для реализации просмотра укладки используется следующая функциональность (компоненты предоставляемые three-js-fiber):

* Canvas – верхний компонент иерархии визуализатора. В него помещаются все компоненты Three.js.
* OrbitControls – позволяет перемещать камеру вокруг центра сцены, а также дает возможность масштабирования.
* AmbientLight – ненаправленное освещение.
* SpotLight – направленное освещение.
* AxesHelper – оси координат.
* Mesh – позволяет отображать полигоны основанные на треугольниках.
* BoxBufferGeometry – создаёт параллелепипед.
* MeshLambertMaterial – добавляет цвет и прозрачность полигонам, находящимся в Mesh.
  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 2.3.
* 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.

Поскольку дерево компонентов не даёт полную информацию о содержании и связях в React приложении, то следует описать функции компонентов, представленных на рис. 12:

* **App** – корень приложения, в него добавляется контекст объектов firebaseFirestore и firebaseAuth.
* **RoutesSwitch** – компонент для роутинга приложения, в нём в зависимости от статуса пользователя определяется какие компоненты-страницы будут доступны, а какие нет.
* **Login** и **Signup** – соответственно компоненты для создания аккаунта и аутнтификации, которые взаимодействуют с firebaseAuth.
* **MainLayout** – главная страница приложения (рис. 16). В компоненте производится вызов функции алгоритма и взаимодействие с firebaseFirestore – записывается результат работы алгоритма. Результаты работы алгоритма также передаются пропсами в PackingView3D и на боковую панель для отображения. Также в этом компоненте созданы функции-коллбеки генерации случайных данных для укладки, загрузки JSON.
* **PackingView3D –** занимает большую часть главной страницы и предоставляет возможность просмотреть укладку с разных сторон. Осуществляет парсинг укладки в компоненты, предоставляемые Three.js, для просмотра 3D.
* **SideMenu –** предоставляет таблицу укладки и возможность открыть главное меню в модальном окне.
* **MainMenu –** основное меню, которое находится в модальном окне, предоставляет доступ к основным функциям приложения. При выборе просмотра статистики вместо кнопок отображается меню с пагинацией. То есть, в данном компоненте также осуществляется доступ к firebaseFirestore. Кроме того, предоставляется возможность скачивания JSON укладки (либо текущей, либо из списка предыдущих).

* 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. 13 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. 14.

* 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.

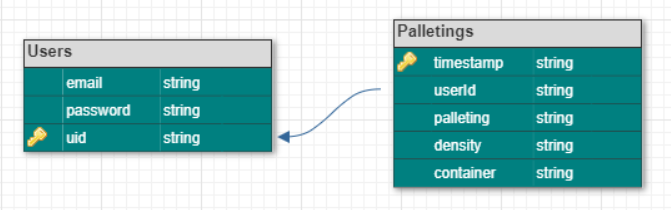


Рис. 13. Схема базы данных

Поскольку React.js придерживается ФП [16], то далее приводятся схемы основных объектов бизнес-логики приложения, в смысле структур данных или моделей данных, а не классов в понимании ООП (рис. 14).

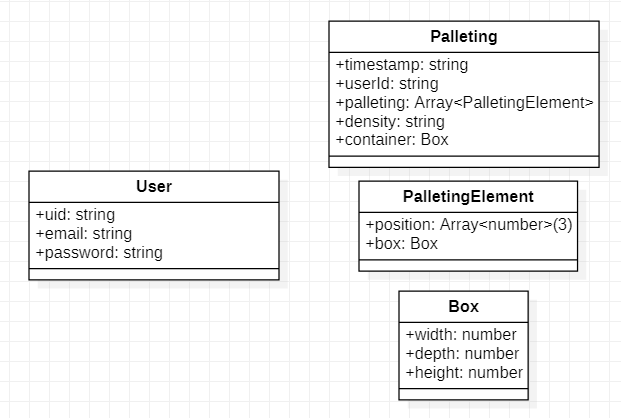


Рис. 14. Схема моделей данных

На рис. 14 можно увидеть схему Palleting, которая описывает документ коллекции Palletings (рис. 13). 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 [21] (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. 15).

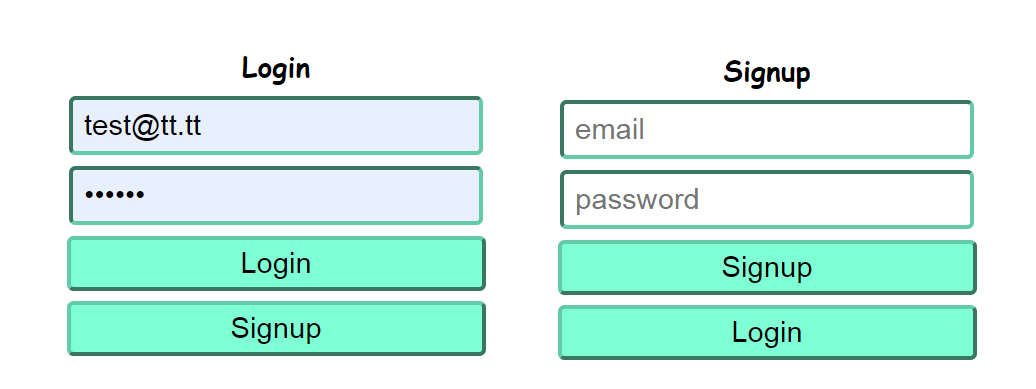


Рис. 15. Формы входа в приложение и создания нового аккаунта

* + 1. ***Журнал учёта укладок***

После аутентификации пользователь видит главную страницу приложения (рис.16). Далее, он может кликнуть по кнопке Menu и увидеть меню приложения (рис 17). В нём пользователь может просмотреть статистику своих укладок и по двойному клику скачать JSON с описанием укладки.

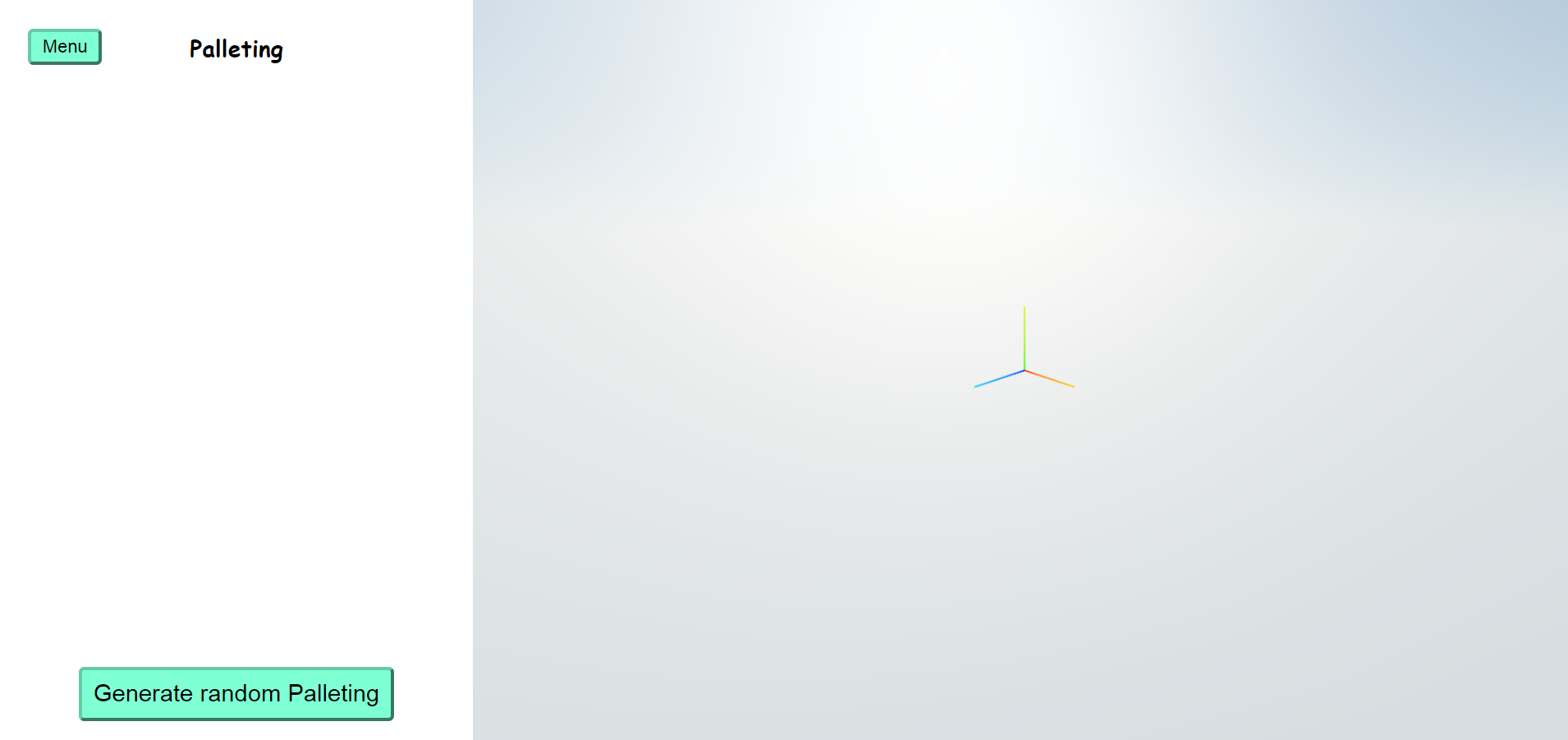


Рис.16. Главная страница приложения



Рис. 17. Меню приложения.

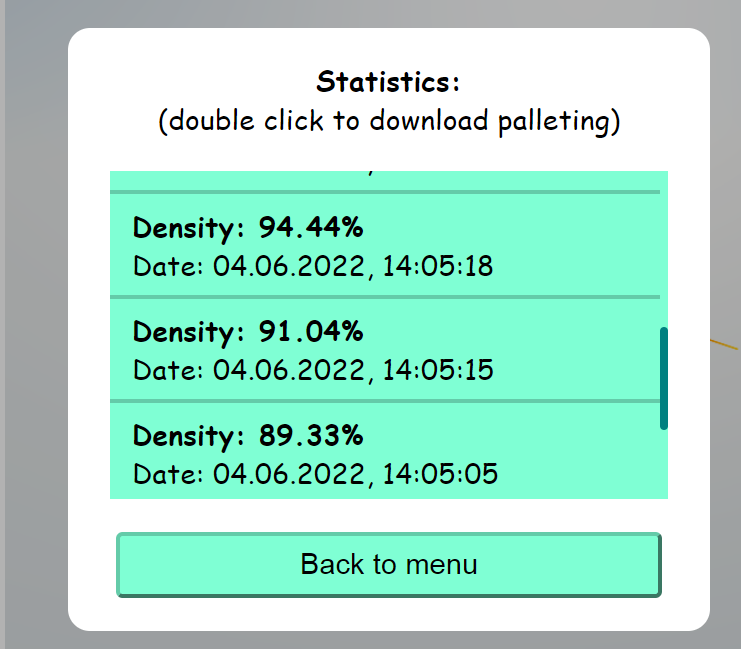


Рис. 18. Список с историей укладок

Статистический список представленный на рис. 18 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.

Стоит отметить способ прослушивания события достижения скролла нижней границы блока, поскольку это нетривиальная задача. Для её решения в конец блока списка, после всех его элементов добавляется блок с высотой и шириной в один пиксель, на который создаётся ссылка с помощью React.useRef. Далее создаётся экземпляр встроенного класса IntersectionObserver и с помощью его метода observe ведётся наблюдение по созданной ссылкой за блоком. Когда скролл пересечёт блок вызовется коллбек с обращением за новой порцией данных.

* 1. ***3D – визуализация укладки***

Важнейшую роль во восприятии приложения пользователем играет визуализация результатов работы алгоритма, особенно в таких случаях как задача укладки. Исходя из этого было решено воспользоваться инструментом Three.js для построения сцены с контейнером и упаковками в нём. Правая часть главной страницы приложения представляет собой компонент Canvas от Three.js. На нём можно увидеть укладку (рис. 19). С помощью инструментов, предоставляемых данной библиотекой возможно поворачивать камеру и осматривать 3D сцену с разных сторон (рис. 20).

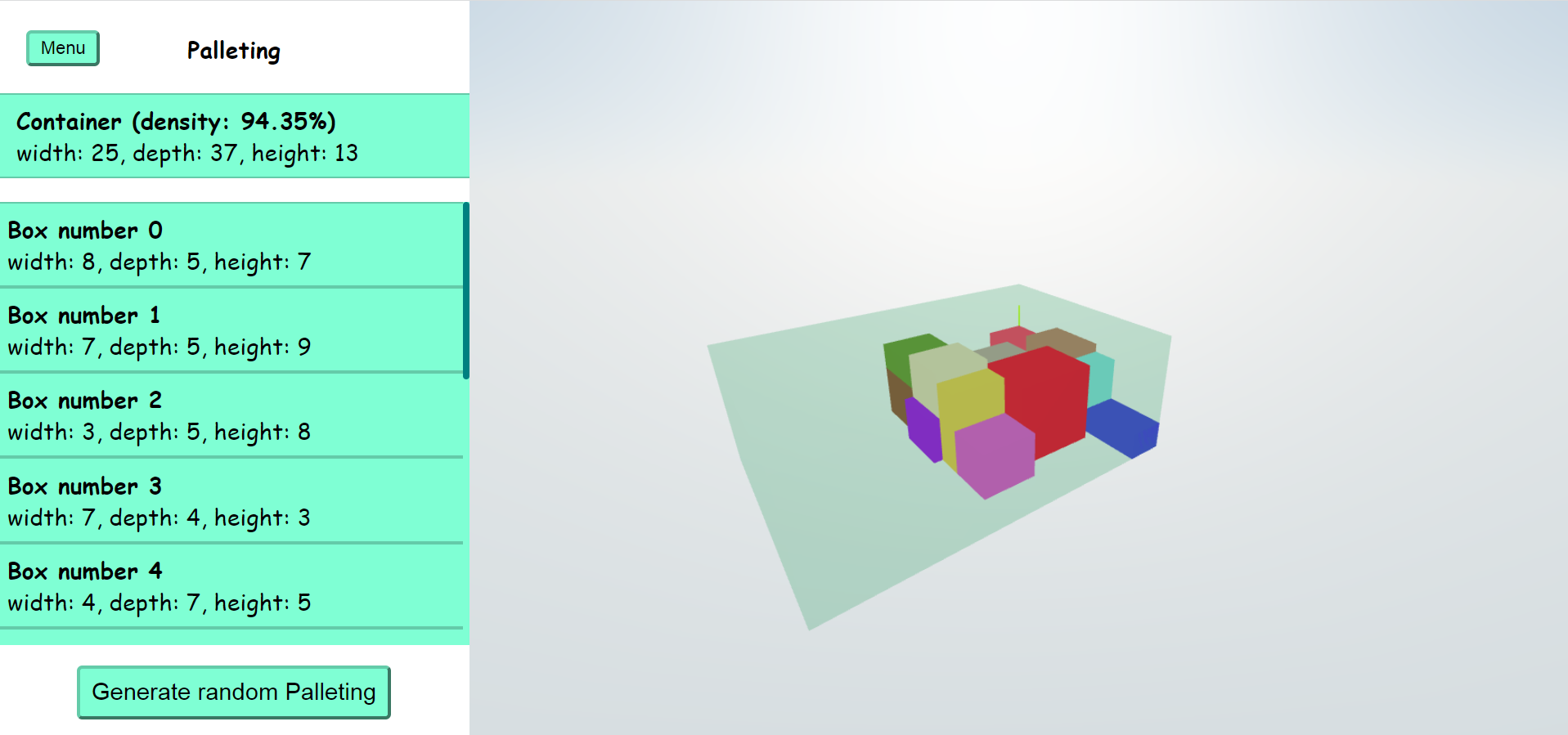


Рис. 19. Главная страница после срабатывания алгоритма укладки

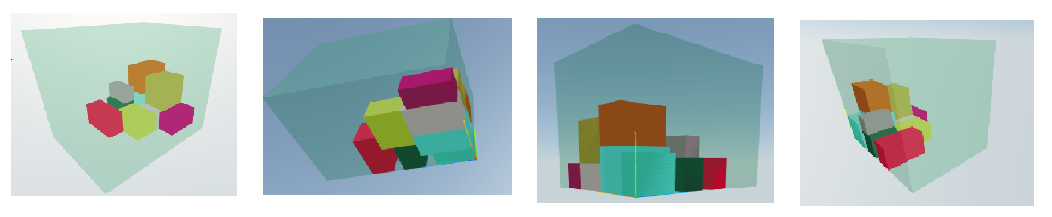


Рис. 20. Укладка с разных ракурсов

Для удобства различия упаковок была разработана функция, генерирующая случайные цвета для каждой коробки в формате hex (шестнадцатеричное представление RGB). Поскольку hex предоставляет FFFFFF16  (= 16 777 12510) цветов, то шанс на одинаковые цвета крайне мал. Также для блоков кроме цвета параметром предусмотрена прозрачность, что важно для наглядности габаритов и в целом видимости контейнера с его содержимым.

* 1. **Реализация алгоритмов в приложении**

Алгоритмы в структуре приложения хранятся в отдельных файлах, из которых они экспортируются. Ключевые слова export и import недоступны в чистом JS без специальных настроек. Посредством инструмента create-react-app в конфигурацию приложения добавляется вышеописанная возможность.

В главном меню приложения (рис. 17) по нажатию одной из двух кнопок:

* Import from JSON чтение из JSON файла входных данных .
* Generate random palleting – генерация случайных входных данных с помощью функции generatePackingData.

вызывается обработчик события onClick, который в свою очередь вызывает коллбэк с вызовом функции укладки, в которую передаются данные, созданные одним из двух вышеназванных способов. В свою очередь, в каждом из импортируемых модулей содержатся функции geneticPacking и generatePackingData соответственно для укладки и генерации данных. Блок-схема функции geneticPacking была представленная ранее, на рис. 4.

**Заключение**

Целью данной выпускной квалификационной работы являлась разработка генетического алгоритма трёхмерной укладки грузов и приложения, которое предоставляет интерфейс для ввода данных укладки, просмотра результатов работы алгоритма, а также позволяющего пользователю просматривать результаты предыдущих укладок.

Во время реализации алгоритма были выполнены следующие задачи:

* Рассмотрены основные эвристические решения задачи трёхмерной укладки;
* На основе рассмотренных эвристических решений составлен алгоритм оптимальный в рамках текущей задачи;
* Оценена оптимальность укладки для реализованного алгоритма с помощью оценки отношения объёмов;
* Оценена оптимальность на выборке случайно сгенерированных данных;
* Реализован генетический алгоритм трёхмерной укладки.

В процессе написания web-приложения были выполнены следующие поставленные задачи:

* Создан интерфейс для загрузки данных о контейнере и упаковках;
* Реализован экспорт данных результата работы алгоритма укладки в формате JSON;
* Создан генератор случайных входных данных для алгоритма. С программным выбором разброса размера контейнеров;
* Добавлена 3D визуализация для просмотра результата работы алгоритма;
* Освоена функциональность Three.js требующаяся для реализации 3D визуализации;
* Реализована авторизация и создание аккаунта;
* Реализовано сохранение результатов работы алгоритма для каждого пользователя;
* Освоена базовая функциональность Firebase (авторизация и операции чтения и добавления в базу данных).

В дальнейшем для увеличения функциональности приложения планируется добавление следующих элементов:

* Создание других вариаций алгоритма укладки, как элемента генетического алгоритма для лучшей оптимизации.
* Создание интерфейса для выбора диапазонов размеров упаковок в случайной генерации для лучших возможностей тестирования.
* Добавление возможности вложенной укладки. То есть сначала будет производится укладка в несколько контейнеров, а затем укладка в более общий контейнер ранее сформированных контейнеров.

**Список использованных источников**

1. Бэнкс А. и др. React и Redux: функциональная веб-разработка. – СПб.: Питер, 2018. – 336 С.
2. Прохоренок Н. Bootstrap и CSS-препроцессор Sass / Н. Прохоренок. СПб.: БХВ-Петербург, 2021. – 496 С.
3. Чеканин В.А, Чеканин А.В. Эффективные модели представления ортогональных ресурсов при решении задачи упаковки // Информационно-управляющие системы. 2012. № 5. С. 29-32.
4. Кныш Д.С., Курейчик В.М. Генетические алгоритмы трассировки коммутационных блоков // Гибридные генетические алгоритмы в задачах автоматизации проектирования. 2011.
5. Чеканин В.А. Эффективное решение задачи двухмерной контейнерной упаковки прямоугольных объектов // Вестник компьютерных и информационных технологий. 2011. № 6. С. 35-39.
6. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
7. Чеканин В.А., Ковшов Е.Е. Моделирование и оптимизация технологических операций в промышленном производстве на основе эволюционных алгоритмов // Технология машиностроения. 2010. № 3. С. 53-57.
8. Филиппова А.С. Моделирование эволюционных алгоритмов решения задач прямоугольной упаковки на базе технологии блочных структур // Информационные технологии. 2006. № 6. Приложение. – 32 с.
9. Курейчик В.М. Алгоритмы одномерной упаковки элементов // Известия ЮФУ. Технические науки. 2013.
10. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
11. . Валиахметова, Ю.И. Мультиметодная технология ортогональной упаковки и ее применение в задачах транспортной логистики / Ю.И. Валиахметова [и др.] // Информационные технологии. – 2009. – № S12. – С. 1-32.
12. Валиахметова, Ю.И. Расширение генетического алгоритма комбинирования эвристик для решения задачи прямоугольной упаковки / Ю.И. Валиахметова, Е.В. Карамова // Вестник ВЭГУ. – 2009. – № 2. – С. 89-94.
13. Введение в UML от создателей языка. 2-е изд. / Г. Буч, Д. Рамбо, И. Якобсон. – М.: ДМК Пресс, 2011. – 496 с.
14. Генетические алгоритмы / Л.А. Гладков, В.В. Курейчик, В.М. Курейчик; под ред. В.М. Курейчика. – 2-е изд., испр. и доп. – М.: ФИЗМАТЛИТ, 2006. – 320 с.
15. Гиля-Зетинов, А.А. Разработка алгоритма укладки паллет на полностью автоматизированном складе / А.А. Гиля-Зетинов, К.К. Панкратов, А.В. Хельвас // Труды МФТИ. – 2019. – Т. 11, № 1. – С. 20-30.
16. Атенсио Луис Функциональное программирование на JavaScript. Как улучшить код JavaScript-программ – Вильямс, 2017. – 304 С.
17. Кузюрин, Н.Н. Проблемы двумерной упаковки и задачи оптимизации в распределенных вычислительных системах / Н.Н. Кузюрин, Д.А. Грушин, С.А. Фомин // Труды Института системного программирования РАН. – 2014. – Т. 26, № 1. – С. 483-501.
18. Головистиков, А.В. Задачи двумерной прямоугольной упаковки и раскроя: обзор / А.В. Головистиков // Информатика. – 2008. – № 4. – С. 18-33.
19. Верхотуров, М.А. Упаковка сложных трёхмерных объектов в прямоугольный контейнер на базе дискретно-логического представления информации / М.А. Верхотуров [и др.] // Известия Самарского научного центра Российской академии наук. – 2014. – Т. 16, № 4–2. – С. 378-383.
20. Документация API Three.js – URL: https://threejs.org/docs/index.html#api/en/geometries/BoxGeometry (дата обращения: 15.05.2021).
21. Документация API Firebase auth – URL: https://firebase.google.com/docs/auth/where-to-start (дата обращения: 29.05.2021).
22. Справочник API React хуков – URL: https://ru.reactjs.org/docs/hooks-reference (дата обращения: 29.04.2021).

**Приложение**

DataGenerator.js

import { randomInteger } from "./generalUtils"

export const **generatePackingData** = () => {

    const container = {

        width: **randomInteger**(20, 25),

        height: **randomInteger**(20, 25),

        depth: **randomInteger**(26, 31)

    }

    const boxesCount = **randomInteger**(10, 15)

    const boxes = []

    for (let i = 0; i < boxesCount; i++) {

        boxes.**push**({

            width: **randomInteger**(3, 9),

            height: **randomInteger**(3, 9),

            depth: **randomInteger**(3, 9)

        })

    }

    return [container, boxes]

}

GeneticPacking.js

import { randomInteger } from "./generalUtils";

const **shuffle** = (array) => {

    for (let i = array.length - 1; i > 0; i--) {

        let j = Math.**floor**(Math.**random**() \* (i + 1));

        [array[i], array[j]] = [array[j], array[i]]

    }

}

const **packing** = (container, boxes, order) => {

    let realVolume = 0

    const result = []

    let currentX = 0

    const **addToResult** = (box, position) => {

        result.**push**({

            position,

            metrics: { width: box.width, height: box.height, depth: box.depth }

        })

    }

    let i = 0

    while (i < boxes.length) {

        let maxDepth = 0

        let sumWidth = 0

        let sumOfAreas = 0

*// ряд вправо*

        while (i < boxes.length && boxes[order[i]].width + sumWidth <= container.width) {

**addToResult**(boxes[order[i]], [sumWidth, 0, currentX])

            if (maxDepth < boxes[order[i]].depth) {

                maxDepth = boxes[order[i]].depth

            }

            sumOfAreas += boxes[order[i]].width \* boxes[order[i]].height

            const basement = boxes[order[i]]

            let sumHeight = basement.height

            i++

*// ряд вверх*

            while (true) {

                if (i < boxes.length

                        && boxes[order[i]].height + sumHeight <= container.height

                        && boxes[order[i]].width <= result[result.length - 1].metrics.width

                        && boxes[order[i]].depth <= result[result.length - 1].metrics.depth

                    ) {

**addToResult**(boxes[order[i]], [sumWidth, sumHeight, currentX])

                    sumHeight += boxes[order[i]].height

                    i++

                } else {

                    if (sumHeight > basement.height) {

                        realVolume += basement.width \* basement.depth \* (sumHeight - basement.height)

                    }

                    break

                }

            }

            sumWidth += basement.width

        }

        if (sumWidth !== 0) {

            realVolume += maxDepth \* sumOfAreas

            currentX += maxDepth

        }

    }

    return [result, realVolume]

}

const **calculateSumOfBoxesVolumes** = (boxes) => {

    let sumOfVolumes = 0

    for (let box of boxes) {

        sumOfVolumes += box.width \* box.depth \* box.height

    }

    return sumOfVolumes

}

const **geneticPacking** = (container, boxes) => {

*// для дальнейшего вычисления критерия подсчитаем сумму объёмов*

    const sumOfVolumes = **calculateSumOfBoxesVolumes**(boxes)

*// генерируем хромосомы*

    let chromosomes = [[]]

    for (let i = 0; i < boxes.length; i++) {

        chromosomes[0].**push**(i)

    }

    for (let i = 1; i < boxes.length; i++) {

        const newPermutation = [...chromosomes[0]]

**shuffle**(newPermutation)

        chromosomes.**push**(newPermutation)

    }

*// элементы укладки и их коэффициент*

    let currentPopulation = []

*// i = количеству поколений*

    for (let i = 0; i < 10000; i++) {

        currentPopulation = []

        for (let chromosome of chromosomes) {

            const [packingValue, volume] = **packing**(container, boxes, chromosome)

            currentPopulation.**push**([packingValue, volume, chromosome])

        }

*// сортируем по убыванию коэффициента*

        currentPopulation.**sort**((a, b) => a[1] - b[1])

*// добавляем лучших*

        const newGenerationChromosomes = [currentPopulation[0][2], currentPopulation[1][2]]

*// скрещиваем не считая две худшие особи*

        for (let j = 0; j < currentPopulation.length - 2; j++) {

*// выбираем три случайные особи*

            const indexOfChromosomes = []

            while (indexOfChromosomes.length !== 3) {

                const index = **randomInteger**(0, currentPopulation.length)

                if (!indexOfChromosomes.**includes**(index)) {

                    indexOfChromosomes.**push**(index)

                }

            }

            const breedingChromosomes = indexOfChromosomes.**map**(index => currentPopulation[index]).**sort**((a, b) => b[1] - a[1])

*// убираем слабую особь*

            breedingChromosomes.**pop**()

*// скрещиваем*

*// выбираем точку скрещивания - элементы до неё идут в новую хромосому*

*// остальные берём из второго родителя (вычеркивая элементы, взятые из первого)*

            const pointOfBreeding = **randomInteger**(0, breedingChromosomes[0].length - 1)

            const firstPart = [...breedingChromosomes[0][2].**slice**(0, pointOfBreeding)]

            const secondPart = [...breedingChromosomes[1][2]]

            for (let elemFromFirstPart of firstPart) {

                const indexOfDeletingElem = secondPart.**findIndex**(elem => elem === elemFromFirstPart)

                if (indexOfDeletingElem >= 0) {

                    secondPart.**splice**(indexOfDeletingElem, 1)

                }

            }

            newGenerationChromosomes.**push**([...firstPart, ...secondPart])

        }

*// мутации*

        for (let j = 2; j < newGenerationChromosomes.length; j++) {

            const randNum = **randomInteger**(0, 100)

            if (randNum === 1) {

                const newPositionOfFirstElem = **randomInteger**(1, newGenerationChromosomes.length - 1)

                let swapContainer = newGenerationChromosomes[j][0]

                newGenerationChromosomes[j][0] = newGenerationChromosomes[j][newPositionOfFirstElem]

                newGenerationChromosomes[j][newPositionOfFirstElem] = swapContainer

            }

        }

        chromosomes = newGenerationChromosomes

    }

    return [currentPopulation[0][0], sumOfVolumes / currentPopulation[0][1]]

}

export default **geneticPacking**

MainLayout.jsx

import { useContext, useState } from 'react';

import { FirebaseContext } from '../..';

import PackingView3D from '../../components/packingView3D/PackingView3D';

import { addPalleting } from '../../components/services/palletingService';

import SideMenu from '../../components/sideMenu/SideMenu';

import { generatePackingData } from '../../helpers/dataGenerator';

import geneticPacking from '../../helpers/geneticPacking';

import styles from './mainLayout.module.scss';

const **MainLayout** = () => {

    const [boxes, **setBoxes**] = **useState**([])

    const [container, **setContainer**] = **useState**({})

    const [density, **setDensity**] = **useState**()

    const {firebaseAuth} = **useContext**(FirebaseContext)

    const **handleRandomizePalleting** = () => {

        const [containerGenerated, boxesGenerated] = **generatePackingData**()

**packing**(boxesGenerated, containerGenerated)

    }

    const **handleUploadJson** = (config) => {

        if (!!config.boxes.length && !!config.container.width) {

**packing**(config.boxes, config.container)

        }

    }

    const **handleStorePalleting** = async (density, container, palleting) => {

        if (density && container && palleting.length && firebaseAuth.currentUser) {

            try {

                const res = await **addPalleting**({

                    density,

                    container,

                    palleting,

                    userId: firebaseAuth.currentUser.uid,

                    timestamp: **Date**.**now**()

                })

            } catch (error) {

                console.**log**(error.message)

            }

        }

    }

    const **packing** = (boxes, container) => {

        const [boxesCalculatedForView, densityCalculated] = **geneticPacking**(container, boxes)

*// const [boxesCalculatedForView, densityCalculated] = simplePacking(container, boxes)*

**handleStorePalleting**(densityCalculated, container, boxesCalculatedForView)

**setContainer**(container)

**setBoxes**(boxesCalculatedForView)

**setDensity**(densityCalculated)

    }

    return (

        <div *className*={styles.container}>

        <header *className*={styles.header}>

        </header>

        <main *className*={styles.main}>

            <div *className*={styles.sideMenu}>

                <**SideMenu**

*blocks*={boxes}

*container*={container}

*density*={density}

*handleGenerate*={**handleRandomizePalleting**}

*handleUploadJson*={**handleUploadJson**}

                />

            </div>

            <div *className*={styles.view}>

                <**PackingView3D** *blocks*={boxes} *containerMetrics*={[container.width, container.height, container.depth]} />

            </div>

        </main>

        <footer *className*={styles.footer}>

        </footer>

        </div>

    )

}

export default **MainLayout**

downloadFile.js

const **downloadJson** = (dataObj, name) => {

    const blob = new **Blob**([JSON.**stringify**(dataObj, null, 2)], {type: 'octet-stream'});

    const href = **URL**.**createObjectURL**(blob);

    const a = **Object**.**assign**(document.**createElement**('a'), {

        href,

        style: 'display: none;',

        download: name

    })

    document.body.**append**(a);

    a.**click**()

**URL**.**revokeObjectURL**(href);

    a.**remove**()

}

export default **downloadJson**

JsonUploader.jsx

import { useRef } from "react"

const **JsonUploader** = ({returnParsedJson}) => {

    const refInput = **useRef**()

    const **handleUploading** = () => {

        refInput.current.**click**()

    }

    const **handleJsonUploaded** = (e) => {

        const fileReader = new **FileReader**()

        fileReader.**readAsText**(e.target.files[0])

        fileReader.**onload** = e => **returnParsedJson**(JSON.**parse**(e.target.result))

        refInput.current.value = null;

    }

    return (

        <>

            <button *onClick*={**handleUploading**}>Import from JSON</button>

            <input *onChange*={**handleJsonUploaded**} *ref*={refInput} *type*='file' *style*={{display: 'none'}} />

        </>

    )

}

export default **JsonUploader**

SideMenu.jsx

import { useState } from 'react';

import MainMenu from '../mainMenu/MainMenu';

import ListItem from '../uiComponents/listItem/ListItem';

import ModalWindow from '../uiComponents/modalWindow/ModalWindow';

import downloadJson from '../utilityComponents/downloadFile';

import styles from './sideMenu.module.scss';

const **SideMenu** = ({container, blocks, density, handleGenerate, handleUploadJson}) => {

    const [isMenuShowed, **setIsMenuShowed**] = **useState**(false);

    const **handleDownload** = () => {

**downloadJson**({container, density, blocks}, 'packing.json');

    }

    return (

        <div *className*={styles.container}>

            <div *className*={styles.header}>

                <button *onClick*={() => **setIsMenuShowed**(true)} *className*={styles.popupMenuButton}>Menu</button>

                <h4 *className*={styles.headerName}>Palleting</h4>

            </div>

            <div *className*={styles.info}>

            { !!container.width &&

                <**ListItem**

*primaryText*={`Container (density: ${(density \* 100).**toFixed**(2)}%)`}

*secondaryText*={`width: ${container.width}, depth: ${container.depth}, height: ${container.height}`}

                />

            }

            { !!blocks.length &&

                <ul *className*={styles.list}>

                    {blocks.**map**((box, ind) => (

                        <**ListItem**

*key*={ind}

*primaryText*={`Box number ${ind}`}

*secondaryText*={`width: ${box.metrics.width}, depth: ${box.metrics.depth}, height: ${box.metrics.height}`}

                        />

                    ))}

                </ul>

            }

            </div>

            <div *className*={styles.footer}>

                <button *onClick*={handleGenerate}>Generate random Palleting</button>

            </div>

            <**ModalWindow** *isShowed*={isMenuShowed} *close*={() => **setIsMenuShowed**(false)}>

                <**MainMenu**

*closeMenu*={() => **setIsMenuShowed**(false)}

*handleUploadJson*={handleUploadJson}

*handleGenerate*={handleGenerate}

*handleDownload*={**handleDownload**}

                />

            </**ModalWindow**>

        </div>

    )

}

export default **SideMenu**

PackingView3D.jsx

import { OrbitControls, Shadow, Sky, Stars } from '@react-three/drei';

import { Canvas } from '@react-three/fiber';

import styles from './packingView3D.module.css';

const **translateBoxPosition** = (metrics, position) => ([

    position[0] + metrics[0] / 2,

    position[1] + metrics[1] / 2,

    position[2] + metrics[2] / 2,

])

const **generateRandomColor** = () => '#' + Math.**floor**(Math.**random**()\*16777215).**toString**(16)

const **Box** = ({position, metrics, color, opacity}) => {

    return (

        <mesh *position*={**translateBoxPosition**(metrics, position)}>

            <**boxBufferGeometry** *attach*='geometry' *args*={metrics}/>

            <**meshLambertMaterial** *attach*='material' *color*={color} *transparent*={!!opacity} *opacity*={opacity}/>

        </mesh>

    )

}

const **PackingView3D** = ({blocks, containerMetrics}) => {

  return (

    <div *className*={styles.container}>

        <**Canvas** *camera*={{position:[50, 25, 50]}}>

            <**OrbitControls** />

            <**Sky** />

            <**ambientLight** *intensity*={0.5} />

            <**spotLight** *position*={[1000, 1000, 1000]} *angle*={0.3} />

            <**axesHelper** *args*={[10]} />

            <**Box** *position*={[0, 0, 0]} *metrics*={containerMetrics} *color*='seagreen' *opacity*={0.25}/>

            {blocks.**map**((block, ind) => (

                <**Box**

*position*={block.position}

*metrics*={[block.metrics.width, block.metrics.height, block.metrics.depth]}

*color*={**generateRandomColor**()}

*key*={ind}

                />

            ))}

        </**Canvas**>

    </div>

  );

}

export default **PackingView3D**;