
Разработка 2D-игр на Java и LibGDX
Практическое руководство
Дмитрий Малышев
dmitry.malyshev@gmail.com
«Замечательное введение в libGDX»
— Mario Zechner, создатель libGDX

Дмитрий Малышев

Разработка 2D-игр на Java и LibGDX

https://litres.ru/74985089
SelfPub; 2026

Аннотация

Практическое руководство для Java-разработчиков, которые хотят создавать 2D-игры на LibGDX. В книге показано, как связать игровой цикл и загрузку ресурсов с графикой, анимацией, управлением, картами Tiled, физикой Box2D, освещением, звуком и интерфейсом. Отдельные главы посвящены оптимизации, пулам объектов и подготовке сборок для компьютеров, Android и браузера. Листинги объясняют, как устроены игровые системы и как применить эти приёмы в собственных проектах. С предисловием Марио Цехнера, создателя LibGDX.

Содержание

О книге 10
О примерах кода 11
Предисловие 12
Благодарности 14
Введение 15
Глава 1. Почему LibGDX 18
1.1. Что такое LibGDX 19
Игра строится в коде 19
Общий код для разных платформ 20
1.2. Возможности и ограничения 22
Как системы работают вместе 22
Что придётся делать самостоятельно 23
Почему стоит начать 23
Глава 2. Создание проекта 25
2.1. Создание проекта и структура модулей 26
Подготовка 26
Создаём проект 27
Где будет жить игра 28
Какую версию Java выбрать 30
Java на Android 30
Пример настройки Java 17 31
Выбор для разных платформ 33
Код и ресурсы 34

2.2. Зависимости и первый запуск 36
Как подключаются библиотеки 36
Запускаем через Gradle Wrapper 37
Наш первый результат на экране 38
Что мы уже получили 40
Глава 3. Игровой цикл и экраны 41
3.1. Жизненный цикл приложения 42
Кто вызывает наши методы 42
Подготовка и работа кадра 43
Приложение свернули — что делать игре 44
Завершение и сохранение 45
3.2. Обновление, отрисовка и delta 47
Время между кадрами 47
Разделяем расчёт и рисование 49
Порядок работы кадра 51
Что делать с большим delta 52
3.3. Game, Screen и переключение экранов 54
Что делает Game 54
Методы экрана 56
Переход из меню в игру 57
Кто освобождает ресурсы 62
Глава 4. Графика и анимация 64
4.1. Текстуры и SpriteBatch 65
Текстура и её область 65
Что делает SpriteBatch 66
Законченный пример 67

Когда удобен Sprite 70
Фильтрация изображения 72
4.2. Порядок отрисовки и прозрачность 73
Что рисуется поверх чего 73
Сортировка по глубине в игре с видом сверху 74
Прозрачность и цвет 77
Обычное и аддитивное смешивание 78
Интерфейс рисуем отдельно 80
Сколько раз batch отправляет данные 81
4.3. Анимация персонажа 83
Подготавливаем кадры 83
Собираем атлас для примера 84
Создаём Animation 86
Время анимации 87
Состояния персонажа 87
Подключаем к экрану 91
Когда сбрасывать таймер 93
Направление и игровые события 94
Глава 5. Ресурсы и экран загрузки 96
5.1. Атласы текстур 97
Как устроен атлас 97
Настройки упаковки 98
Подготавливаем общий атлас игры 99
Получаем изображение по имени 100
Как разделять ресурсы 102

Кто освобождает атлас 103
5.2. Загрузка ресурсов через AssetManager 105
Подготавливаем очередь 105
Обрабатываем загрузку по кадрам 107
Получаем готовые ресурсы 109
Не превращаем загрузку в блокирующее ожидание 110
5.3. Экран загрузки 112
Минимум Scene2D для этого примера 112
Ресурсы самого экрана 113
Запуск игры 114
Фон и интерфейс 116
Готовность и ошибка загрузки 121
Переход в игру 121
5.4. Освобождение ресурсов 125
Владелец и пользователь ресурса 125
Ресурсы из AssetManager 128
Что считает менеджер 129
Общие ресурсы и ресурсы локации 130
Экран и ресурс — разные времена жизни 130
Ресурсы, загруженные напрямую 132
Глава 6. Шрифты в LibGDX 133
6.1. TrueType и BitmapFont 134
Что именно мы загружаем 134
Загружаем готовый растровый шрифт 135
Создаём BitmapFont из TrueType 137

Почему отдельно задаём символы 141
Генерация в памяти и экспорт — разные операции 142
Освобождение ресурсов 143
6.2. Свой FontGenerator 144
Что делает этот класс 144
Почему шрифт нужно подготавливать белым 145
Реализация FontGenerator 146
Использование 150
Создаём один раз и используем повторно 151
Кто освобождает шрифты 153
6.3. Размеры и качество текста 155
Размер глифов и размер на экране 155
Начинаем с нескольких ролей текста 156
Фильтрация и масштабирование 157
Измеряем текст, а не угадываем ширину 158
Подбор разрешения под viewport и HiDPI 159
Не генерируем шрифт при каждом движении границы окна 162
6.4. Шрифты в браузере, текстурные страницы и память 164
Что меняется в GWT-сборке 164
Экспортируем шрифт через Hiero 165
Загружаем готовые шрифты через AssetManager 166

Если способы подготовки отличаются по платформам 168
Почему несколько шрифтов могут занимать много памяти 169
Упаковываем два размера в общие страницы 170
Как страницы влияют на renderCalls 175
Глава 7. Камера в 2D-игре 177
7.1. Координаты игрового мира и экрана 178
Координаты мира 178
Координаты нажатия 179
Из экрана в игровой мир: unproject() 179
Проверяем попадание в область viewport 181
Из мира на экран: project() 182
Мир и интерфейс 183
7.2. Камера и Viewport 185
OrthographicCamera 185
FitViewport: сохраняем заданную область 186
Обновление при изменении окна 188
Отрисовка через камеру 189
ExtendViewport: показываем больше мира 190
Проверяем настройку 193
Конец ознакомительного фрагмента. 194

Дмитрий Малышев
Разработка 2D-игр на Java и LibGDX

Малышев Дмитрий Разработка 2D-игр на Java и LibGDX. Практическое руководство. 2026.

О книге

Практическое руководство для Java-разработчиков, которые хотят создавать 2D-игры на LibGDX. В книге показано, как настройка проекта и игровой цикл связаны с графикой, анимацией, вводом, картами Tiled, физикой Box2D, освещением, частицами, звуком и интерфейсом. Также рассматриваются владение ресурсами и их освобождение, производительность, пулы объектов и подготовка сборок для компьютера, Android и браузера.

Большинство примеров посвящено играм с видом сверху. Предполагается знание классов, интерфейсов и коллекций Java. Предварительный опыт работы с LibGDX не требуется.

О примерах кода

Код в книге приведён, чтобы объяснить основные приёмы, детали использования API и настройки проекта. В листингах может быть опущен вспомогательный код, а некоторые классы и ситуации условны и служат для пояснения. Вместе примеры не образуют полный готовый к запуску проект.

© Дмитрий Малышев, 2026

dmitry.malyshev@gmail.com

Предисловие

В далёком 2009 году я захотел разрабатывать игры для Android. Эмулятор Android тогда работал медленно как черепаха, а необходимость загружать приложение на телефон после каждого крошечного изменения очень быстро начала утомлять. Поэтому я поступил так, как поступает любой программист, столкнувшийся с небольшим неудобством: написал собственный фреймворк. Идея была простой: пишешь игру один раз, запускаешь её на компьютере, пока разрабатываешь и дорабатываешь, а когда всё готово — выпускаешь на телефоне. Эта маленькая хитрость со временем превратилась в libGDX.

Я совершенно не ожидал того, что произошло дальше. Всего за пару лет libGDX стал одним из самых популярных фреймворков для разработки игр под Android. Люди создавали с его помощью вещи, о которых я даже мечтать не мог: от небольших игр, сделанных за выходные на игровых джемах, до таких проектов, как Ingress и Slay the Spire. В 2016 году я передал управление замечательной команде основных разработчиков, которые продолжают поддерживать и развивать фреймворк по сей день.

libGDX никогда не диктует вам, как именно создавать свою игру. Нет редактора, в котором вы обязаны работать, нет архитектуры, которой вы должны подчиняться. Только

набор строительных блоков и полная свобода действий — а вместе с ней и достаточно возможностей попасть в неприятности. Это прекрасно, если вы знаете, что делаете. А если нет… что ж, свободы у вас всё равно предостаточно.

И вот здесь на помощь приходит эта книга. Дмитрий знакомит вас со всем, что необходимо практически любой 2D-игре: графикой, анимацией, обработкой ввода, физикой, освещением и способами добиться достаточной производительности. Но что ещё важнее — он показывает, как все эти компоненты объединяются в настоящую игру.

Мой совет: не ограничивайтесь чтением. Набирайте код самостоятельно. Ломайте его. Меняйте значения и смотрите, что произойдёт. Создайте маленькую ужасную игру, а затем ещё одну — уже чуть менее ужасную. Именно так я сам учился, когда в детстве мой Game Boy заставил меня задуматься, как вообще работают видеоигры. И с тех пор я так и не нашёл лучшего способа научиться их создавать.

А когда вы что-нибудь сделаете — покажите это другим. Сообщество libGDX всегда проявляло себя с лучшей стороны, когда люди делились тем, что создали, и знаниями, которые получили в процессе.

А теперь идите и создавайте игры!

Марио Цехнер (Mario Zechner)

Создатель libGDX

Благодарности

Я хочу выразить искреннюю благодарность Марио Цехнеру, создателю libGDX, за его добрые слова, поддержку и за то, что он написал предисловие к этой книге.

Особую благодарность я хочу выразить Натану Свиту, одному из создателей libGDX, за то, что он нашёл время познакомиться с книгой, поделился вдумчивыми замечаниями и дал ценные технические рекомендации. Его советы помогли мне улучшить материал и задуматься о направлениях дальнейшей работы.

Я глубоко признателен Марио и Нейту не только за помощь с этой книгой, но и за создание и развитие libGDX — фреймворка, благодаря которому я многие годы учился, экспериментировал и получал удовольствие от разработки игр.

Я также благодарен всему сообществу libGDX за то, что оно поддерживает и развивает фреймворк, делится знаниями и вдохновляет разработчиков по всему миру.

Дмитрий Малышев

Введение

В какой момент набор изображений становится игрой? Когда персонаж начинает двигаться? Когда появляются препятствия и цель? Или когда камера, свет и звук заставляют почувствовать, что перед нами живой мир?

Обычно это происходит постепенно. Сначала мы выводим изображение, затем добавляем управление, загружаем карту и настраиваем движение. Но особенно интересно становится тогда, когда отдельные системы начинают работать вместе: камера открывает пространство впереди, факел освещает стену, а за препятствием появляется тень.

Эта книга — о том, как создавать такие сцены с помощью LibGDX и понимать, что происходит внутри.

Мы пройдём путь от создания проекта и игрового цикла до графики, управления, карт, физики, освещения и интерфейса. Разберём загрузку ресурсов, частицы и звук, научимся искать причины тормозов и подготовим проект к сборке и публикации. При этом будем смотреть не только на отдельные классы, но и на связь между ними: кто обновляет объект, кто рисует его и кто освобождает использованные ресурсы.

Для кого эта книга

Книга рассчитана на разработчиков, знакомых с Java: классами, интерфейсами и коллекциями. Предварительный опыт работы с LibGDX не требуется.

Большинство примеров связано с 2D-играми с видом сверху. Мы разберём, как организовать сцену, связать её системы и подготовить игру к запуску на целевых устройствах. Если вы уже работаете над своей игрой, отдельные разделы можно использовать для решения конкретных задач.

Как работать с примерами

При первом знакомстве лучше читать книгу последовательно: масштаб координат влияет на физику и свет, камера — на обработку касаний, а порядок отрисовки — на внешний вид всей сцены.

Код в книге приведён, чтобы передать суть приёмов, показать особенности API и необходимые настройки. Здесь есть короткие фрагменты и законченные классы, самостоятельные примеры и продолжения предыдущих. Они не составляют последовательный листинг одного готового проекта. Некоторые ситуации и вспомогательные классы условны: они нужны, чтобы разобрать конкретный нюанс. Полный проект со всеми ресурсами, настройками и связующим кодом не является задачей этой книги. При переносе примера в свою игру проверяйте используемые координаты, зависимости, порядок вызовов и освобождения ресурсов.

Самый полезный способ чтения — сразу проверять идеи. Добавьте один источник света, измените его радиус, поставьте перед ним стену. Переключите тип Viewport и измените размер окна. Сравните управление одним и двумя джойстиками. Такие небольшие опыты помогают понять решение

лучше, чем простое копирование кода.

Наша цель — получить основу для собственной игры и научиться уверенно её развивать. Какой будет эта игра, какие правила в ней появятся и чем она запомнится игрокам — решать вам.

Глава 1. Почему LibGDX

Прежде чем создавать проект, разберёмся, какие задачи берёт на себя LibGDX, что остаётся разработчику и как устроена работа с фреймворком.

План главы

- 1.1. Что такое LibGDX
- 1.2. Возможности и ограничения

1.1. Что такое LibGDX

LibGDX — это фреймворк для разработки игр на Java. Он предоставляет готовые средства для работы с графикой, звуком, вводом и другими системами, которые нужны игре.

Например, чтобы нарисовать персонажа, не требуется самостоятельно организовывать передачу каждого изображения видеокарте. Мы загружаем текстуру и используем средства отрисовки LibGDX. Чтобы получить положение пальца на экране или воспроизвести выстрел, обращаемся к соответствующему API.

При этом правила игры остаются нашим кодом. LibGDX не знает, как должен двигаться персонаж, кого атаковать противнику и что произойдёт после попадания пули. Эти решения принимает разработчик.

Игра строится в коде

В стандартном рабочем процессе LibGDX нет единого визуального редактора, в котором мы собираем всю игру из готовых сцен. Основная работа происходит в среде разработки: мы пишем классы, связываем системы и запускаем приложение.

Это не означает, что всё приходится делать вручную. Кар-

ты можно создавать в Tiled, текстуры упаковывать в атласы, а эффекты настраивать в редакторе частиц. Затем игра загружает подготовленные данные и использует их.

Получается понятное разделение: **редакторы помогают создавать содержимое, а код определяет, как оно работает в игре.**

Например, в Tiled мы рисуем карту и отмечаем стены, точки появления персонажей и места для факелов. В программе читаем эти объекты, создаём препятствия, размещаем персонажей и включаем освещение.

Общий код для разных платформ

LibGDX поддерживает Windows, Linux, macOS, Android, iOS и браузер. Основную логику можно размещать в общем модуле проекта, а запуск приложения и особенности конкретной платформы — в отдельных модулях.

Так, расчёт здоровья, движение противников и работа оружия могут использовать один и тот же код. А управление будет различаться: на компьютере — клавиатура и мышь, на телефоне — касания и виртуальные джойстики.

Это удобно для разработки: основную игру можно запускать на компьютере, быстро проверять изменения, а затем тестировать мобильную или браузерную сборку.

Однако общий код не отменяет различий между устройствами. Размер экрана, доступная производительность, по-

ведение звука и особенности ввода всё равно требуют проверки на целевой платформе.

1.2. Возможности и ограничения

LibGDX предоставляет основу: отрисовку, ввод, звук, загрузку ресурсов и инструменты для интерфейса. Box2D поможет работать со столкновениями, а box2dLight — со светом и тенями. Поведение игры складывается из совместной работы этих систем.

Как системы работают вместе

Возьмём факел на стене. Его изображение рисуется вместе с окружением, частицы создают пламя, источник света освещает соседние объекты, а звук дополняет эффект. Эти элементы нужно разместить в одной точке, обновлять во время игры и убрать при выгрузке уровня.

На таком примере хорошо видно, как устроена работа с LibGDX: отдельные инструменты предоставляют библиотеки, а их совместное поведение организуем мы.

Связь эффекта с объектом зависит от задачи. Свет фонарика перемещается вместе с персонажем, а след снаряда остаётся там, где снаряд пролетел. Звук запускаем в момент нужного игрового события.

Что придётся делать самостоятельно

LibGDX предоставляет инструменты, но организацию игры выбирает разработчик. Нам нужно решить, как создаются и удаляются объекты, кто обновляет их состояние, как обрабатываются попадания и кому принадлежат ресурсы.

Это требует аккуратности. Если смешать все задачи в одном классе, развитие проекта быстро станет трудным. Поэтому будем разделять ответственность между системами, сохраняя код достаточно простым для понимания.

Есть и технические ограничения: устройство располагает конечным объёмом памяти и времени на обработку кадра. Сотни эффектов или источников света нельзя считать бесплатными. Мы разберём, как находить дорогие операции, переиспользовать объекты через пулы и выбирать настройки, которые дают хороший результат без лишней нагрузки.

Цель — понимать, за счёт чего работает решение и где проходит граница его применения. Тогда его можно уверенно изменять под свою задачу.

Почему стоит начать

Если вы знаете основы Java, у вас уже есть база для работы с LibGDX. Привычные классы, методы и коллекции станут инструментами для создания персонажей, уровней и иг-

ровых механик. А результат каждого шага можно будет увидеть на экране.

Не требуется заранее спроектировать большую игру. Начать можно с небольшой сцены, в которой персонаж двигается, сталкивается со стенами и стреляет. Такая сцена позволит проверить идею, почувствовать управление и понять, что хочется развивать дальше.

По мере чтения у вас будет складываться набор решений: как загрузить уровень, настроить обзор, добавить факел, собрать интерфейс или избежать лишних выделений памяти. Их можно сочетать по-разному — в зависимости от того, какую игру вы задумали.

Начнём с первого работающего проекта. В следующей главе создадим его, разберём структуру и запустим приложение.

Глава 2. Создание проекта

План главы

- 2.1. Создание проекта и структура модулей
- 2.2. Зависимости и первый запуск

2.1. Создание проекта и структура модулей

Первый шаг — получить проект, который можно собрать и запустить. После этого у нас будет место для всех будущих систем: персонажа, карты, камеры и оружия.

Для создания проекта воспользуемся **GDX-Liftoff** — генератором, который подготовит модули, настройки сборки и стартовые классы. Собрать такую структуру вручную возможно, но на первом этапе удобнее начать с готовой основы.

Подготовка

Нам понадобятся JDK 21 и среда разработки — IntelliJ IDEA или Android Studio. JDK нужен для запуска инструментов сборки и компиляции Java-кода. После установки проверяем команды java -version и javac -version: обе должны показывать версию 21. В IDE также выбираем JDK 21 для запуска Gradle: терминал и среда разработки могут использовать разные установленные JDK.

Для начала создадим настольную версию игры. Её удобно запускать во время разработки: изменили код, проверили результат и продолжили работу. Android и браузер можно добавить, когда понадобится проверять игру на этих платформах.

Если сразу выбираем Android, потребуется также Android SDK. Его можно установить через SDK Manager в Android Studio.

Создаём проект

Скачиваем файл .jar со страницы релизов GDX-Liftoff[1] и запускаем его. Если двойной щелчок не открывает программу, используем терминал:

```
java -jar gdx-liftoff-VERSION.jar
```

[1] GDX-Liftoff: https://github.com/libgdx/gdx-liftoff/releases

Вместо VERSION указываем версию в имени скачанного файла. Команду выполняем из папки, в которой он находится.

В генераторе задаём основные параметры:

Параметр	Пример	Назначение
Project Name	`LibGDXBook`	Название проекта
Package	`com.example.libgdxbook`	Базовый Java-пакет
Main Class	`BookGame`	Главный класс игры
Project Path	Пустая папка для проекта	Место сохранения файлов

Выбираем Java, настольную платформу и простой шаблон приложения. Модуль core обязателен — в нём будет находиться общий код игры. Дополнительные расширения пока оставляем выключенными: физику и освещение подключим тогда, когда дойдём до соответствующих задач.

В этой редакции ориентируемся на LibGDX 1.14.2. В поле Java Version выбираем 8 — это уровень языка для исходного кода, а не версия установленного JDK. Сборку запускаем на JDK 21. Причины такого выбора и варианты для других платформ разберём ниже.

Нажимаем кнопку генерации и открываем созданную папку как Gradle- проект в IDE. При первом импорте понадобится интернет: система сборки скачает необходимые зависимости.

Где будет жить игра

После генерации файлов может показаться много. Но для начала достаточно понять назначение нескольких элементов:

Элемент	Что в нём находится
`core`	Общая логика и отрисовка игры
`lwjgl3`	Запуск и настройки настольного приложения
`assets`	Изображения, карты, звуки и другие ресурсы
`build.gradle`	Настройки сборки и зависимостей; есть и у отдельных модулей
`settings.gradle`	Список подключённых модулей
`gradle.properties`	Общие параметры проекта
`gradlew`, `gradlew.bat`	Скрипты запуска Gradle Wrapper

Если выбрать дополнительные платформы, появятся модули android, html или ios.

Большую часть времени мы будем работать в core. Здесь появятся персонажи, правила движения, игровые экраны, камера и обработка попаданий.

Модуль lwjgl3 содержит стартовый класс настольной версии, обычно Lwjgl3Launcher. Он создаёт приложение, задаёт параметры окна и передаёт управление главному классу игры.

Такое разделение позволяет запускать общую игровую логику через разные платформенные оболочки. Например, поведение пули остаётся в core, а настройка окна относится к настольному модулю.

Какую версию Java выбрать

JDK для сборки, уровень языка и доступные Java API — разные вещи. JDK запускает Gradle и предоставляет компилятор. Уровень языка определяет, какие конструкции разрешены в исходном коде, а доступные API зависят от целевой платформы. Поэтому установка JDK 21 не означает, что общий игровой код обязательно нужно писать на Java 21.

С LibGDX можно работать с кодом на Java 8, 11, 17 и 21, но выбор зависит от платформ и инструментов сборки. Для новых проектов GDX-Liftoff рекомендует JDK 21[2], в том числе для удобной упаковки настольных сборок. Запуск Gradle на JDK 21 должен поддерживаться версией Gradle в проекте; используем Wrapper, подготовленный генератором.

Java на Android

Современные Android-проекты не ограничены языковым уровнем Java 8. При подходящих версиях Android Gradle Plugin и D8/R8 можно использовать конструкции Java 17, в том числе:

- var для локальных переменных — Java 10;
- выражения switch — Java 14;
- текстовые блоки — Java 15;
- record и сопоставление с образцом в instanceof — Java

16.

Часть этих конструкций обрабатывает сам Java-компилятор. Для возможностей, требующих дополнительного преобразования, Android-

[2] GDX-Liftoff рекомендует JDK 21: https://github.com/libgdx/gdx-liftoff

инструменты используют **desugaring**: D8/R8 преобразуют код при подготовке DEX для Android Runtime (ART).

Поддержка синтаксиса не означает доступность всей стандартной библиотеки Java SE. Например, использование java.time или java.nio.file зависит от compileSdk, минимальной версии Android и настройки Core Library Desugaring. Этот механизм предоставляет поддерживаемое подмножество API на старых устройствах, а не переносит на Android всю библиотеку установленного JDK. Подробности и список API приведены в документации Android[3].

JDK 21 можно использовать для сборки Android-проекта с уровнем кода Java 17. Переход самого кода на Java 21 требует отдельной проверки возможностей Android Gradle Plugin, D8/R8 и используемых API: одной смены номера версии недостаточно.

Пример настройки Java 17

Следующие настройки — **альтернатива для проекта**

под Android и desktop, если мы решили использовать Java 17. Для повторения примеров книги менять выбранный уровень Java 8 не требуется.

В существующем блоке android файла android/build.gradle обновляем compileOptions:

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_17
        targetCompatibility JavaVersion.VERSION_17
    }
}
```

В core/build.gradle согласуем уровень общего кода:

```
java {
    sourceCompatibility = JavaVersion.VERSION_17
    targetCompatibility = JavaVersion.VERSION_17
}
```

Если генератор уже задал эти параметры в другом месте, обновляем существующую настройку, не оставляя конфликтующих значений. sourceCompatibility задаёт уровень языка, а targetCompatibility — версию формата Java-классов; сами по себе они не обеспечивают доступность API на

[3] документации Android: https://developer.android.com/

studio/write/java8-support

Android. Эти различия разобраны в документации о версиях Java в Android- сборках[4].

В таком проекте Gradle запускается на JDK 21, компилятор создаёт классы с уровнем Java 17, а Android-инструменты преобразуют их в DEX. **Устанавливать Java 17 или 21 на телефон не нужно:** приложение выполняется в Android Runtime.

Выбор для разных платформ

Для начала можно пользоваться такими ориентирами. Это практические варианты настройки, а не полный перечень поддерживаемых сочетаний.

Целевые платформы	Уровень общего Java-кода	JDK для сборки
Только Android	Java 17 с учётом доступных API	JDK 21
Android и desktop	Java 17 с учётом ограничений Android	JDK 21
Android, desktop и HTML через GWT	Java 8 или 11 для базовой конфигурации GDX-Liftoff	JDK 21
Android, desktop и iOS через RoboVM	Java 8 с отдельной проверкой доступных API	JDK 21; для iOS нужны macOS и Xcode
Только desktop	Java 21	JDK 21

Для браузера важно учитывать версию GWT. Генератор GDX-Liftoff рекомендует уровни 8 или 11 для своей

GWT-конфигурации. При этом **сам GWT начиная с 2.12 поддерживает конструкции Java 12–17**, поэтому Java 17 нельзя считать запрещённой для GWT вообще. Её использование требует совместимых версий backend и сборочных инструментов, соответствующей настройки компилятора и проверки эмулируемых Java API. Возможности описаны в примечаниях к выпускам GWT[5].

Для RoboVM уровень языка Java 8 также не гарантирует наличия всех API Java 8: общий код проверяем по возможностям этой платформы.

В книге сохраняем Java 8 для исходного кода и JDK 21 для сборки. Это выбранная основа для переносимых примеров; она не означает, что современные Android-проекты обязаны ограничиваться Java 8.

[4] документации о версиях Java в Android-сборках: https://developer.android.com/build/jdks

[5] примечаниях к выпускам GWT: https://www.gwtproject.org/release-notes.html#Release_Notes_2_12_0

Код и ресурсы

Java-классы размещаем в каталогах исходного кода соответствующих модулей, а содержимое игры — в assets. По мере работы там появятся папки images, maps, audio и другие.

Путь к ресурсу внутри программы задаём относительно

assets. Например, файл assets/images/player.png будет загружаться по пути images/player.png. Имя папки assets в этот путь не входит.

Не стоит копировать общую игровую логику в платформенные модули. Если персонаж должен одинаково двигаться на компьютере и телефоне, правила движения принадлежат core. Различаться будут источники команд — клавиатура или касания.

Основа готова: теперь мы знаем, где писать код и хранить ресурсы. В следующем разделе разберём зависимости и запустим приложение — впервые увидим результат нашего проекта на экране.

2.2. Зависимости и первый запуск

Пока наш проект — это набор файлов. Первый запуск превратит его в работающее приложение: появится окно, начнётся отрисовка, а наш код получит управление.

Сначала разберём, откуда берутся библиотеки и как запускается сборка. Затем заменим стартовый пример небольшим собственным приложением.

Как подключаются библиотеки

Зависимость — это библиотека, которую использует проект. Gradle скачивает её из указанного репозитория и добавляет к сборке. Вручную искать и копировать файлы .jar обычно не требуется.

В core/build.gradle уже есть подключение LibGDX. Например, оно может выглядеть так:

```
dependencies {
    api "com.badlogicgames.gdx:gdx:$gdxVersion"
}
```

Здесь com.badlogicgames.gdx — группа, gdx — название библиотеки, а gdxVersion — переменная с её версией. В проекте GDX-Liftoff она обычно задаётся в gradle.properties.

Этот фрагмент уже подготовлен генератором — добавлять его второй раз не нужно. По тому же принципу позже подключим физику и другие расширения.

У настольного модуля свои зависимости: библиотека запуска LWJGL3, общий модуль core и необходимые нативные библиотеки. Нативные библиотеки содержат код для конкретной платформы, используемый, например, при работе с графикой.

Поэтому для некоторых расширений одного подключения в core недостаточно: требуются и платформенные зависимости. Когда дойдём до Box2D, разберём полный набор.

После изменения файлов сборки обновляем Gradle-проект в IDE, чтобы она получила новые зависимости.

Запускаем через Gradle Wrapper

В корне проекта находятся gradlew и gradlew.bat. Они запускают **Gradle Wrapper**, который использует версию Gradle, заданную проектом. Отдельно устанавливать Gradle не требуется.

Открываем терминал в папке с settings.gradle и выполняем команду.

В Windows PowerShell:

```
.\gradlew.bat lwjgl3:run
```

В Linux или macOS:

```
./gradlew lwjgl3:run
```

Команда означает: выполнить задачу run модуля lwjgl3. Эту же задачу можно запустить из окна Gradle в IDE.

Если в Linux или macOS появляется Permission denied, разрешаем выполнение скрипта командой chmod +x gradlew и повторяем запуск.

Первый запуск может занять больше времени из-за скачивания компонентов сборки и библиотек. После успешной сборки откроется окно со стартовым примером выбранного шаблона.

Если Gradle сообщает, что модуль lwjgl3 не найден, проверяем две вещи: выбран ли корень нужного проекта и присутствует ли этот модуль в settings.gradle.

Наш первый результат на экране

Закрываем окно приложения. Теперь откроем главный класс BookGame в модуле core и полностью заменим его содержимое:

```java
package com.example.libgdxbook;

import com.badlogic.gdx.ApplicationAdapter;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.utils.ScreenUtils;

public class BookGame extends ApplicationAdapter {
    private SpriteBatch batch;
    private BitmapFont font;

    @Override
    public void create() {
        batch = new SpriteBatch();
        font = new BitmapFont();
    }

    @Override
    public void resize(int width, int height) {
        if (width <= 0 || height <= 0) {
            return;
        }
        batch.getProjectionMatrix().setToOrtho2D( 0, 0, width, height );
    }

    @Override
    public void render() {
        ScreenUtils.clear(0.06f, 0.08f, 0.12f, 1f);

        batch.begin();
        font.draw(batch, "Hello, LibGDX!", 40, 80);
        batch.end();
    }

    @Override
    public void dispose() {
        font.dispose();
        batch.dispose();
    }
}
```

Пакет должен совпадать с тем, который мы указали в генераторе. Стартовый класс Lwjgl3Launcher уже создаёт BookGame, поэтому менять механизм запуска не требуется.

Снова выполняем lwjgl3:run. Получаем тёмное окно с надписью **Hello, LibGDX!**

В примере create() подготавливает объекты, render() рисует кадр, а dispose() освобождает ресурсы. Метод resize() задаёт координаты отрисовки по размеру окна: здесь одна единица соответствует одному логическому пикселю окна.

Пока надпись размещена в точке (40, 80). Подробно координаты, камеру и адаптацию изображения к разным экранам разберём в главе 7.

Используем английский текст, поскольку стандартный шрифт этого примера не содержит кириллицу. Шрифты для русского интерфейса подготовим отдельно.

Что мы уже получили

У нас есть приложение, которое собирается, открывает окно и выполняет наш код при каждом кадре. Можно изменить цвет фона или положение надписи и сразу увидеть результат.

На этой основе дальше появятся изображения, персонаж и игровой мир. Но прежде разберём, когда LibGDX вызывает наши методы и как организовать обновление игры, чтобы её скорость не зависела от частоты кадров.

Глава 3. Игровой цикл и экраны

План главы

- 3.1. Жизненный цикл приложения
- 3.2. Обновление, отрисовка и delta
- 3.3. Game, Screen и переключение экранов

3.1. Жизненный цикл приложения

Игрок запустил игру, изменил размер окна, свернул приложение и вернулся через минуту. Для нас это разные события: нужно подготовить ресурсы, перестроить изображение, сохранить состояние и корректно продолжить работу.

LibGDX сообщает о таких событиях через **методы жизненного цикла**. Мы уже использовали некоторые из них в первом примере. Теперь разберём, какую задачу решает каждый.

Кто вызывает наши методы

Главный класс игры реализует интерфейс ApplicationListener. Платформенная часть LibGDX вызывает его методы в нужные моменты.

Удобнее наследоваться от ApplicationAdapter: он содержит пустые реализации всех методов интерфейса. Мы переопределяем только те, которые нужны приложению.

Метод	Назначение
`create()`	Начальная подготовка приложения
`resize(width, height)`	Реакция на изменение размера области отображения
`render()`	Обновление игры и отрисовка очередного кадра
`pause()`	Обработка приостановки приложения
`resume()`	Обработка возвращения к работе
`dispose()`	Освобождение ресурсов при штатном завершении

При запуске вызывается create(), затем resize() с начальным размером. После этого приложение получает повторяющиеся вызовы render().

Собственный бесконечный цикл внутри render() писать не нужно. Один вызов должен выполнить работу для одного кадра и вернуть управление LibGDX.

Подготовка и работа кадра

В create() создаём необходимые объекты и запускаем начальную загрузку. Ресурсы, связанные с графическим контекстом, например Texture и SpriteBatch, подготавливаем здесь, а не в конструкторе главного класса: во время создания экземпляра игры графическая среда ещё может быть не готова.

Метод render() будет вызываться многократно. Здесь считываем команды игрока, обновляем состояние мира и рисуем результат.

Например, текстуру персонажа загружаем один раз при подготовке, а затем используем при отрисовке каждого кадра. Повторная загрузка внутри render() создаст лишнюю работу и расход памяти.

В resize() обновляем параметры отображения. Позже будем передавать новые размеры в Viewport, чтобы игровая сцена и интерфейс правильно размещались при изменении окна или ориентации устройства.

Приложение свернули — что делать игре

На телефоне игрок может перейти в другое приложение прямо во время боя. В pause() стоит сохранить важные данные и сбросить временное состояние управления: удерживаемую стрельбу, направление джойстика и захваченные касания.

Иначе после возвращения персонаж может продолжить движение, хотя палец уже давно убран с экрана.

В resume() восстанавливаем необходимое состояние. Автоматически продолжать бой необязательно: можно оставить игровую паузу и дождаться команды игрока.

Пауза приложения и пауза игрового процесса — разные вещи. Первую сообщает платформа, вторую вводим мы. Например, меню паузы может остановить движение противников, пока render() продолжает рисовать интерфейс.

Условия вызова pause() и resume() зависят от платформы и настроек приложения. Поэтому потерю фокуса окна или переключение браузерной вкладки нужно проверять отдельно, а не считать универсальным сигналом игровой паузы.

Завершение и сохранение

В dispose() освобождаем ресурсы, которыми владеет приложение. В нашем первом примере это шрифт и SpriteBatch. После освобождения использовать их для отрисовки нельзя.

Однако **сохранение прогресса не стоит откладывать только до dispose()**. Операционная система может завершить процесс без штатного вызова этого метода. Важные изменения сохраняем по ходу игры, а при приостановке фиксируем актуальное состояние.

Возвращение из фона не всегда означает вызов resume(): если процесс был завершён, приложение создастся заново. При новом запуске загружаем сохранённые данные и восстанавливаем игровые объекты. Ссылки на прежние объекты этого не обеспечивают.

Теперь у каждого действия есть своё место: подготовка — в create(), работа кадра — в render(), реакция на размеры — в resize(), обработка приостановки — в pause() и resume(), освобождение ресурсов — в dispose().

В следующем разделе разберём сам кадр: как разделить обновление и отрисовку и почему персонаж должен двигать-

ся с одинаковой скоростью при разном FPS.

3.2. Обновление, отрисовка и delta

Представим, что персонаж за каждый кадр сдвигается на две единицы. При 30 FPS он пройдёт 60 единиц за секунду, а при 120 FPS — уже 240. Получается, что на более быстром компьютере игра тоже работает быстрее.

Для большинства игровых действий это неправильно. Скорость персонажа, полёт пули и длительность анимации должны определяться временем, а не количеством нарисованных кадров.

Время между кадрами

LibGDX позволяет получить время между кадрами в секундах:

```
float delta = Gdx.graphics.getDeltaTime();
```

При 60 FPS оно составляет примерно 0.0167 секунды, при 30 FPS — 0.0333. Значение меняется: один кадр может потребовать больше времени, другой — меньше.

Если скорость задана в единицах в секунду, перемещение за кадр рассчитывается так:

```
x += speed * delta;
```

Например, при скорости 120 единиц в секунду объект сдвинется примерно на 2 единицы при 60 FPS и на 4 — при 30 FPS. За одинаковое время он пройдёт одинаковое расстояние.

Мы задаём скорость в секунду, а delta переводит её в перемещение за текущий кадр.

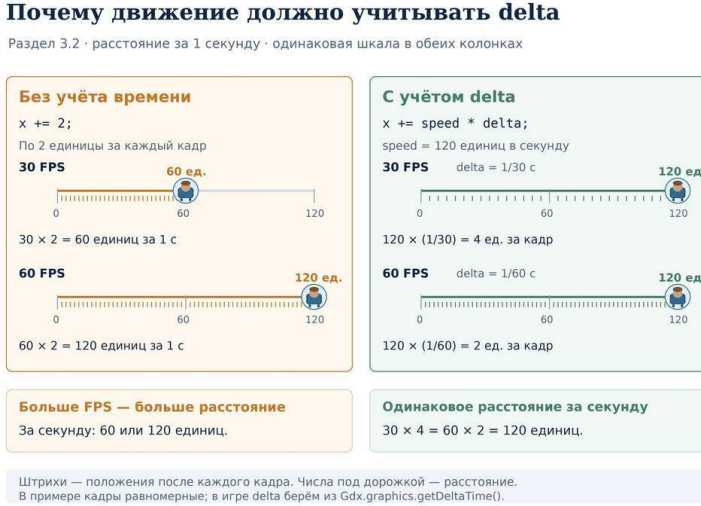


Рисунок 3.1. Перемещение за одну секунду при 30 и 60 FPS.

Учёт delta сохраняет скорость

120 единиц в секунду; кадры в каждом режиме равномерные.

Это убирает прямую зависимость скорости от FPS. Однако точность столкновений и сложных расчётов всё ещё зависит от размера шага — к этому вернёмся ниже.

Разделяем расчёт и рисование

В render() удобно выделить два действия: обновить состояние игры и нарисовать полученный результат.

Продолжим пример BookGame. Добавим импорты:

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Input;
```

В классе объявим поля:

```
private static final float SPEED = 120f;
private float x = 40f;
```

Затем заменим render() и добавим два вспомогательных метода:

```java
@Override
public void render() {
    float delta = Gdx.graphics.getDeltaTime();

    update(delta);
    draw();
}

private void update(float delta) {
    float direction = 0f;

    if (Gdx.input.isKeyPressed(Input.Keys.LEFT)) {
        direction -= 1f;
    }
    if (Gdx.input.isKeyPressed(Input.Keys.RIGHT)) {
        direction += 1f;
    }

    x += direction * SPEED * delta;
}

private void draw() {
    ScreenUtils.clear(0.06f, 0.08f, 0.12f, 1f);

    batch.begin();
    font.draw(batch, "Hello, LibGDX!", x, 80f);
    batch.end();
}
```

Методы create(), resize() и dispose() оставляем прежними.

Теперь стрелки перемещают надпись влево и вправо. При одновременном нажатии обеих клавиш направления ком-

пенсируются.

Границы движения пока не ограничены: надпись может уйти за окно. Методы update() и draw() мы создали сами; LibGDX вызывает render(), а тот обращается к ним.

В update() меняется координата, а draw() только использует её. Такое разделение позволяет остановить игровой мир, сохранив отрисовку: например, показать меню паузы поверх неподвижной сцены.

Порядок работы кадра

Когда систем станет больше, порядок их обновления приобретёт значение. Для нашей игры удобно придерживаться такой последовательности: 1. Получить команды управления.

2. Обновить движение и игровую логику, выполнить шаг физики, если она используется. 3. Обработать результаты столкновений и синхронизировать игровые объекты. 4. Обновить анимации, эффекты и камеру. 5. Нарисовать мир и интерфейс.

Если камера обновляется до движения персонажа, она будет следовать за его предыдущим положением. Если свет привязан к персонажу, его координаты тоже должны соответствовать уже обновлённому состоянию.

Конкретный порядок зависит от устройства игры. Главное — понимать, какие данные получает каждая система и успе-

ли ли они обновиться.

Одно и то же действие не должно случайно выполняться дважды за кадр. Если эффект частиц уже обновлён через update(delta), дополнительное обновление при отрисовке ускорит его.

Что делать с большим delta

После задержки или остановки в отладчике время кадра может оказаться большим. Объект тогда переместится сразу на значительное расстояние, а простая проверка пересечения в конечной точке может пропустить препятствие.

Для локальной игры иногда ограничивают шаг:

```
float delta = Math.min( Gdx.graphics.getDeltaTime(), 0.1f );
```

Это уменьшает скачок, но имеет цену: часть прошедшего времени отбрасывается. При постоянных задержках игровое время начнёт отставать от реального. Для таймеров, которые должны учитывать настоящее время, такая замена не подходит.

Ограничение шага само по себе не предотвращает прохождение через тонкую стену. А срок контракта, действующего и при закрытой игре, определяем по сохранённому времени окончания, а не по сумме ограниченных шагов.

Более устойчивый подход для физики — **фиксирован-**

ный шаг. Прошедшее время накапливается, а симуляция выполняется небольшими одинаковыми шагами, например по 1 / 60 секунды. Между двумя отрисовками может потребоваться несколько таких обновлений.

Фиксированный шаг не гарантирует одинаковый FPS и требует ограничения объёма работы при перегрузке. Его практическую реализацию разберём в главе о Box2D.

Теперь наш кадр устроен понятно: сначала меняем состояние, затем показываем результат, а движение рассчитываем через время. В следующем разделе разделим приложение на экраны — меню, загрузку и саму игру.

3.3. Game, Screen и переключение экранов

У игры появляется меню, затем экран загрузки, а после него — игровой мир. У каждого свои действия и отрисовка. Если разместить всё в одном render(), он быстро обрастёт проверками: какое состояние сейчас активно и что нужно обновлять.

LibGDX предлагает для этого пару **Game и Screen**. Главный класс управляет активным экраном, а экран отвечает за свой участок приложения.

Что делает Game

Game реализует ApplicationListener и передаёт вызовы render(), resize(), pause() и resume() текущему экрану.

Для переключения используется:

```
setScreen(new MenuScreen(this));
```

При этом у прежнего экрана вызывается hide(), у нового — show(), а затем resize() с текущими размерами.

Обычный setScreen() не освобождает прежний экран. Мы сами решаем, сохранить его для возвращения

или вызвать dispose().

Если переопределяем render() в наследнике Game, для передачи отрисовки текущему экрану вызываем super.render(). Аналогично сохраняем передачу событий при переопределении resize(), pause() и resume(). В примере ниже используем унаследованные реализации.



Рисунок 3.2. Переход между экранами и порядок вызовов hide(), show() и resize(). Прежний экран освобождает его владелец.

Методы экрана

Экран реализует интерфейс Screen. Для коротких примеров удобно наследоваться от ScreenAdapter, который предоставляет пустые реализации его методов.

Метод	Для чего используется
`show()`	Активация экрана: например, подключение обработчика ввода
`render(float delta)`	Обновление и отрисовка экрана
`resize(width, height)`	Обновление камеры и области отображения
`pause()`, `resume()`	Обработка приостановки приложения и возвращения
`hide()`	Отключение экрана
`dispose()`	Окончательное освобождение принадлежащих ему ресурсов

В отличие от главного класса приложения, у Screen нет метода create(). Подготовку можно выполнять в конструкторе, если экран создаётся при работающем приложении, или при первом show().

Важно помнить: show() может вызываться повторно, если тот же экземпляр снова становится активным. Не нужно каждый раз загружать одни и те же ресурсы без освобождения предыдущих.

Переход из меню в игру

Сделаем два экрана: меню с приглашением нажать Enter и игровую сцену с возвратом по Escape. Так мы проверим переключение без дополнительных изображений и кнопок.

Заменим BookGame следующим кодом. Все три класса размещаем в отдельных файлах одного пакета com.example.libgdxbook.

```java
package com.example.libgdxbook;

import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Screen;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.utils.ScreenUtils;

public class BookGame extends Game {
    private SpriteBatch batch;
    private BitmapFont font;

    @Override
    public void create() {
        batch = new SpriteBatch();
        font = new BitmapFont();
        setScreen(new MenuScreen(this));
    }

    public void replaceScreen(Screen next) {
        Screen previous = getScreen();
        if (previous == next) {
            return;
        }

        setScreen(next);

        if (previous != null) {
            previous.dispose();
        }
    }

    public void drawText(String text) {
```

BookGame — продолжение листинга

```
        if (Gdx.graphics.getWidth() <= 0
                || Gdx.graphics.getHeight() <= 0) {
            return;
        }
        ScreenUtils.clear(0.06f, 0.08f, 0.12f, 1f);

        batch.getProjectionMatrix().setToOrtho2D( 0, 0,
            Gdx.graphics.getWidth(),
            Gdx.graphics.getHeight()
        );

        batch.begin();
        font.draw(batch, text, 40f, 80f);
        batch.end();
    }

    @Override
    public void dispose() {
        Screen current = getScreen();
        setScreen(null);

        if (current != null) {
            current.dispose();
        }

        font.dispose();
        batch.dispose();
    }
}
```

Здесь replaceScreen() — наш вспомогательный метод. Он

переключает экран и освобождает предыдущий. Такой порядок подходит для примера, в котором прежний экран больше не используется.

Общую отрисовку надписи вынесли в drawText(), чтобы не повторять её в двух классах. Пока используем координаты окна; позже у игровых экранов появятся собственные камеры и Viewport.

Создадим MenuScreen.java:

```
package com.example.libgdxbook;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Input;
import com.badlogic.gdx.ScreenAdapter;

public class MenuScreen extends ScreenAdapter {
    private final BookGame game;

    public MenuScreen(BookGame game) {
        this.game = game;
    }
```

MenuScreen — продолжение листинга

```
    @Override
    public void render(float delta) {
        if (Gdx.input.isKeyJustPressed(Input.Keys.ENTER)) {
            game.replaceScreen(new GameScreen(game));
            return;
        }

        game.drawText("Press ENTER to start");
    }
}
```

И GameScreen.java:

```
package com.example.libgdxbook;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.Input;
import com.badlogic.gdx.ScreenAdapter;

public class GameScreen extends ScreenAdapter {
    private final BookGame game;

    public GameScreen(BookGame game) {
        this.game = game;
    }

    @Override
    public void render(float delta) {
        if (Gdx.input.isKeyJustPressed(Input.Keys.ESCAPE)) {
            game.replaceScreen(new MenuScreen(game));
            return;
        }

        game.drawText("Game screen. ESC to return");
    }
}
```

Запускаем приложение. Enter открывает игровую сцену, Escape возвращает меню. Используем isKeyJustPressed(), чтобы реагировать на новое нажатие, а не каждый кадр удержания.

После переключения сразу выходим из метода через return: прежний экран уже отключён и освобождён, продолжать его работу не следует.

Новый экран получит render() на следующем кадре. Каждый переход в этом примере создаёт новый экземпляр: возврат из меню начинает новую сцену, а не восстанавливает предыдущую.

Кто освобождает ресурсы

В примере SpriteBatch и шрифт принадлежат BookGame. Экраны пользуются ими, но не вызывают для них dispose().

Если экран создаёт собственную текстуру, музыку или Stage, он отвечает за их освобождение. Для ресурсов, загруженных через общий AssetManager, правила владения разберём в главе 5.

hide() означает «экран больше не активен», а dispose() — «его ресурсы больше не нужны». Это разные события.

Стандартный Game.dispose() вызывает у текущего экрана только hide(). Поэтому в BookGame.dispose() очистка вы-

полнена явно: сначала setScreen(null) отключает экран, затем освобождаются его ресурсы и общие объекты отрисовки.

Например, при открытии отдельного экрана инвентаря можно сохранить игровой экран и затем вернуться к тому же экземпляру. Наш replaceScreen() для этого не подходит: он предназначен для окончательной замены. Сохранённые экраны нужно учитывать отдельно и освободить, когда они действительно перестанут использоваться.

Если экран подключает InputProcessor, при переключении также передаём ввод новому экрану. Сам setScreen() не управляет обработчиками ввода.

В этой главе мы разобрали основу работы приложения: жизненный цикл, обновление через время и разделение на экраны. Теперь у будущих систем есть понятное место — можно добавлять графику, ресурсы и игровой мир, сохраняя порядок в коде.

Глава 4. Графика и анимация

План главы

- 4.1. Текстуры и SpriteBatch
- 4.2. Порядок отрисовки и прозрачность
- 4.3. Анимация персонажа

4.1. Текстуры и SpriteBatch

Чтобы нарисовать персонажа, предмет или фон, нужно загрузить изображение и разместить его в игровом мире. В LibGDX для этого обычно используются Texture, TextureRegion и SpriteBatch.

Текстура и её область

Texture загружает изображение и создаёт текстуру, доступную для отрисовки видеокартой:

```
Texture texture = new Texture( Gdx.files.internal("images/player.png") );
TextureRegion playerRegion = new TextureRegion(texture);
```

Файл должен находиться в каталоге ресурсов приложения. Если этот каталог называется assets, полный путь к файлу будет assets/images/player.png, но в Gdx.files.internal() передаём путь без assets/.

В этом примере playerRegion охватывает всё изображение. Если на одной текстуре расположено несколько кадров или объектов, можно выбрать прямоугольный фрагмент:

```
TextureRegion frame = new TextureRegion( texture, 0, 0, 32, 32 );
```

Здесь выбирается область размером 32 × 32 пикселя из левого верхнего угла исходной картинки. Она должна помещаться в текстуре. Эти координаты относятся к изображению, а не к положению персонажа в игровом мире.

TextureRegion хранит ссылку на текстуру и границы области, а не отдельную копию пикселей. Несколько областей могут использовать одну текстуру; отдельный dispose() для них не нужен. В разделе 5.1 получим такие области из атласа по именам.

Что делает SpriteBatch

SpriteBatch накапливает данные изображений и отправляет их на видеокарту порциями. Поэтому несколько вызовов draw() могут попасть в одну отправку.

Отрисовка выполняется между begin() и end():

```
batch.begin();
batch.draw(playerRegion, 100f, 80f, 64f, 64f);
batch.draw(playerRegion, 200f, 80f, 64f, 64f);
batch.end();
```

Первые два числа после playerRegion — положение нижнего левого угла изображения, следующие два — ширина и высота в единицах игрового мира. Размер исходной картинки и размер объекта при отрисовке могут различаться. Если

их пропорции не совпадают, изображение растянется.

Не вызываем begin() и end() для каждого персонажа. Обычно в одном проходе рисуем фон, предметы и персонажей через одну камеру. Это не гарантирует одну отправку на видеокарту: смена текстуры или заполнение внутреннего буфера могут вызвать дополнительные отправки. Подробнее разберём это в разделе 4.2.

Перед отрисовкой через другую камеру или другой инструмент завершаем текущий проход и настраиваем следующий. Например, вызовы ShapeRenderer не должны вклиниваться между SpriteBatch.begin() и end().

Законченный пример

Создадим самостоятельный GraphicsScreen, используя устройство экранов из раздела 3.3. Поместим квадратное изображение персонажа в images/player.png внутри каталога ресурсов. На экране оно займёт область 64 × 64 в центре игрового пространства 480 × 270.

Для настройки отображения используем FitViewport. Пока достаточно знать, что он сохраняет заданные пропорции и не растягивает персонажа при изменении окна. Если пропорции окна отличаются, по краям остаются полосы. Подробно камеру и viewport рассмотрим в разделе 7.2.

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.ScreenAdapter;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.utils.viewport.FitViewport;

public class GraphicsScreen extends ScreenAdapter {
    private final SpriteBatch batch = new SpriteBatch();
    private final Texture texture = new Texture(
        Gdx.files.internal("images/player.png")
    );
    private final TextureRegion playerRegion =
        new TextureRegion(texture);
    private final FitViewport viewport = new FitViewport(480f, 270f);
```

GraphicsScreen — продолжение листинга

```
    @Override
    public void render(float delta) {
        if (Gdx.graphics.getWidth() <= 0
                || Gdx.graphics.getHeight() <= 0) {
            return;
        }

        Gdx.gl.glClearColor(0.12f, 0.14f, 0.16f, 1f);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        viewport.apply();

        batch.setProjectionMatrix(viewport.getCamera().combined);
        batch.begin();
        batch.draw(playerRegion, 208f, 103f, 64f, 64f);
        batch.end();
    }

    @Override
    public void resize(int width, int height) {
        if (width <= 0 || height <= 0) {
            return;
        }

        viewport.update(width, height, true);
    }

    @Override
    public void dispose() {
        batch.dispose();
        texture.dispose();
    }
}
```

Размещаем класс в пакете нашего проекта. Экран создаём после инициализации приложения, когда графический контекст уже доступен. Например, чтобы открыть его из BookGame вместо меню, используем вспомогательный метод из раздела 3.3:

```
replaceScreen(new GraphicsScreen());
```

Он переключает экран и освобождает предыдущий. Если прежний экран нужно сохранить для возвращения, вместо окончательной замены используем обычный setScreen() и отдельно учитываем время жизни сохранённого экрана.

При переключении Game вызывает у нового экрана resize() с текущими размерами окна. В нашем методе viewport.update(..., true) настраивает отображение и центрирует камеру. Координаты (208, 103) помещают изображение размером 64 × 64 в центр области 480 × 270.

Перед рисованием viewport.apply() задаёт экранную область вывода, а setProjectionMatrix() передаёт batch матрицу камеры. Экран очищается каждый кадр, после чего персонаж рисуется заново.

В этом самостоятельном примере текстура и batch принадлежат GraphicsScreen, поэтому он освобождает их в dispose(). Общие ресурсы могут иметь другого владельца; эту схему разберём в разделе 5.4. Правило переключения и освобождения экранов остаётся тем же, что в разделе 3.3.

Когда удобен Sprite

Если изображению нужны собственные положение, раз-

мер, поворот и цвет, эти свойства можно хранить в Sprite из пакета com.badlogic.gdx.graphics.g2d:

```
Sprite sprite = new Sprite(playerRegion);
sprite.setBounds(208f, 103f, 64f, 64f);
sprite.setOriginCenter();
sprite.setRotation(30f);
```

Создаём его один раз при подготовке экрана. В открытом проходе batch рисуем:

```
sprite.draw(batch);
```

Это альтернативный способ показать то же изображение. Если используем его в GraphicsScreen, заменяем прежний batch.draw(playerRegion, ...), чтобы не нарисовать персонажа дважды.

setOriginCenter() задаёт центр вращения. Положение спрайта по-прежнему описывает нижний левый угол его прямоугольника до поворота.

Sprite удобен для графического представления, но не обязан быть самим игровым объектом. Положение персонажа можно хранить в игровой модели, а перед отрисовкой передавать спрайту. Создание ещё одного Sprite на основе той же

области не создаёт новую текстуру.

Фильтрация изображения

При масштабировании картинки можно сохранить резкие пиксели или сгладить переходы. Например, для резких границ задаём:

```
texture.setFilter( Texture.TextureFilter.Nearest,
    Texture.TextureFilter.Nearest
);
```

Nearest часто используют для пиксельной графики, а Linear — для сглаженного масштабирования. Первый параметр задаёт фильтр уменьшения, второй — увеличения. Настраиваем их при подготовке текстуры; фильтрация действует на все её области.

Резкие пиксели сами по себе не гарантируют ровное движение: для пиксельной графики также важны масштаб камеры и размещение объектов. Пока выберем подходящий фильтр и проверим изображение при разных размерах окна.

Теперь мы умеем загрузить изображение и показать его в игровом мире. В следующем разделе разберём порядок слоёв, прозрачность и цвет, чтобы персонажи и окружение правильно перекрывали друг друга.

4.2. Порядок отрисовки и прозрачность

Правильно загрузить текстуры недостаточно. Персонаж может оказаться под полом, дым — закрыть интерфейс, а после вспышки остальные предметы вдруг начать светиться. Такие ошибки часто связаны с порядком отрисовки и настройками SpriteBatch.

Что рисуется поверх чего

При обычной 2D-отрисовке через SpriteBatch без проверки глубины позднее нарисованные изображения перекрывают предыдущие. Batch не определяет самостоятельно, где находится земля, а где персонаж.

Простой порядок для сцены: 1. Земля и фон. 2. Декорации под персонажами. 3. Персонажи и объекты, которые могут перекрывать друг друга. 4. Эффекты переднего плана. 5. Интерфейс.

В следующих фрагментах SpriteBatch и камера уже настроены, как в разделе 4.1. groundRegion, playerRegion и foregroundRegion — загруженные области земли, персонажа и переднего плана:

```
batch.begin();
batch.draw(groundRegion, 0f, 0f, 480f, 270f);
batch.draw(playerRegion, 208f, 103f, 64f, 64f);
batch.draw(foregroundRegion, 180f, 120f, 120f, 100f);
batch.end();
```

Передний план рисуется последним и перекрывает персонажа там, где его пиксели непрозрачны. Координата y сама по себе не меняет этот порядок: его задаёт последовательность вызовов draw().

Сортировка по глубине в игре с видом сверху

Если персонаж проходит перед деревом, он должен закрывать его основание. Если проходит за ним — дерево должно закрывать персонажа. Для такого взаимного перекрытия фиксированного порядка слоёв недостаточно.

Обычно сортируем объекты по вертикальной координате точки опоры: стопам персонажа или основанию дерева. При системе координат, где Y растёт вверх, объекты с большим Y рисуем раньше. Тогда персонаж, стоящий ниже дерева, окажется поверх него.

Создадим упрощённое графическое представление объекта. TextureRegion и SpriteBatch находятся в пакете com.badlogic.gdx.graphics.g2d:

```
public class SceneObject {
    public final TextureRegion region;
    public float x;
    public float y; // точка опоры: низ изображения по центру
    public float width;
    public float height;

    public SceneObject(TextureRegion region,
            float x, float y, float width, float height) {
        this.region = region;
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public void draw(SpriteBatch batch) {
        batch.draw(region, x - width / 2f, y, width, height);
    }
}
```

В игровом экране объявим список и компаратор. Используем Array из com.badlogic.gdx.utils и Comparator из java.util:

```
private final Array<SceneObject> sceneObjects = new Array<>();
private final Comparator<SceneObject> depthOrder =
    (a, b) -> Float.compare(b.y, a.y);
```

Добавляем графические представления персонажей и деревьев в sceneObjects при подготовке сцены. После обновления их позиций сортируем список и рисуем его между фоном и передним планом:

```
sceneObjects.sort(depthOrder);
```

Код к разделу 4.2 — продолжение листинга

```
batch.begin();
batch.draw(groundRegion, 0f, 0f, 480f, 270f);
for (int i = 0; i < sceneObjects.size; i++) {
    sceneObjects.get(i).draw(batch);
}
batch.draw(foregroundRegion, 180f, 120f, 120f, 100f);
batch.end();
```

Этот проход заменяет предыдущий пример. Персонажа, уже добавленного в список, отдельно второй раз не рисуем.

Персонажи и деревья должны находиться в общем сортируемом списке, если требуется их взаимное перекрытие. Сортировка только персонажей не изменит порядок относительно деревьев, нарисованных отдельным проходом.

В примере нижний край изображения совпадает с точкой опоры. Если под ногами есть прозрачный отступ, учитываем его при размещении картинки, но сортируем по настоящему положению ног. Для мостов, этажей и сложных построек могут понадобиться дополнительные слои или разделение изображения на части.

sceneObjects здесь отвечает за отрисовку. Если игровая

логика использует собственный список объектов, не меняем порядок её обновления ради перекрытия картинок. Организацию этих списков разберём в разделе 10.2.

Прозрачность и цвет

PNG может содержать альфа-канал: прозрачные пиксели оставляют видимым ранее нарисованный фон. В стандартном SpriteBatch смешивание включено по умолчанию.

Дополнительную прозрачность при вызове batch.draw() задаём цветом batch:

```
batch.begin();
batch.setColor(1f, 1f, 1f, 0.5f);
batch.draw(playerRegion, 208f, 103f, 64f, 64f);
batch.setColor(Color.WHITE);
batch.end();
```

Здесь Color — класс из com.badlogic.gdx.graphics. Первые три компонента умножают цвет изображения, четвёртый — его альфу. Белый цвет сохраняет исходные цвета, а альфа 0.5f уменьшает непрозрачность каждого пикселя вдвое. Полностью прозрачные участки остаются прозрачными.

Таким же способом можно придать изображению оттенок. Например, batch.setColor(1f, 0.5f, 0.5f, 1f) уменьшает зелёную и синюю составляющие, создавая красноватую окраску.

Это умножение цвета, поэтому оно не позволяет произвольно перекрасить любые исходные пиксели.

begin() и end() не возвращают цвет batch к белому. После отдельного эффекта восстанавливаем цвет, который ожидает следующая отрисовка. В наших примерах это Color.WHITE.

У Sprite из раздела 4.1 есть собственный цвет. Если рисуем через sprite.draw(batch), задаём прозрачность самому спрайту:

```
sprite.setAlpha(0.5f);
sprite.draw(batch); // внутри открытого прохода batch
```

Для окраски используем sprite.setColor(...). Цвет этого спрайта сохраняется между кадрами, пока мы не изменим его снова. Пример с batch.setColor() относится к обычному batch.draw(playerRegion, ...); при Sprite.draw() используется цвет, записанный в самом спрайте.

Обычное и аддитивное смешивание

Смешивание определяет, как цвет нового изображения соединяется с уже нарисованным фоном. Для обычных изображений с непреумноженным альфа- каналом используем стандартную комбинацию:

```
batch.setBlendFunction( GL20.GL_SRC_ALPHA, GL20.GL_ONE_MINUS_SRC_ALPHA );
```

Полупрозрачный пиксель участвует в результате вместе с фоном: чем выше его альфа, тем сильнее он закрывает фон. GL20 импортируем из com.badlogic.gdx.graphics.

Для некоторых светящихся эффектов подходит аддитивное смешивание: цвет эффекта добавляется к уже нарисованному изображению.

```
batch.begin();
batch.setBlendFunction(GL20.GL_SRC_ALPHA, GL20.GL_ONE);
batch.draw(glowRegion, 190f, 85f, 100f, 100f);

batch.setBlendFunction( GL20.GL_SRC_ALPHA, GL20.GL_ONE_MINUS_SRC_ALPHA );
batch.end();
```

glowRegion — заранее подготовленная область свечения. Этот режим подходит для вспышек, искр и магических эффектов. Для дыма обычно нужно обычное смешивание: дым должен закрывать фон, а не только осветлять его.

После эффекта восстанавливаем обычный режим. Как и цвет, выбранная комбинация смешивания сохраняется для последующих вызовов. За настройками особенно важно следить после частиц, которые могут менять режим внутри своего прохода; этот случай разберём в разделе 13.2.

Приведённые настройки рассчитаны на графику с непреумноженным альфа- каналом. Если ресурсы подготовлены с преумноженной альфой, нужна соответствующая комбина-

ция смешивания — менять её без учёта подготовки изображения нельзя.

Интерфейс рисуем отдельно

Игровой мир и интерфейс обычно используют разные камеры. Перемещение камеры мира не должно уводить за экран кнопки и надписи. Поэтому сначала завершаем отрисовку мира, затем применяем viewport интерфейса и рисуем его.

В LibGDX для интерфейса удобно использовать Stage из Scene2D. Первый пример с ним появится на экране загрузки в разделе 5.3, а устройство игрового интерфейса разберём в главе 14.

Если uiStage уже создан, перед его отрисовкой вызываем uiStage.getViewport().apply(), затем — uiStage.draw(). Последний метод сам открывает и завершает проход своего batch. Не вызываем его внутри незавершённого прохода отрисовки мира, особенно если обе системы используют один batch.

Когда добавим освещение, оно тоже займёт своё место в последовательности. Интерфейс обычно рисуем после света, чтобы затемнение мира не влияло на кнопки и текст. Подключение света к кадру рассмотрим в разделе 12.2.

Сколько раз batch отправляет данные

Количество вызовов draw() не равно количеству отправок геометрии на видеокарту. Сразу после завершения прохода через batch.end() можно прочитать:

```
int calls = batch.renderCalls;
```

Это число вызовов отрисовки с момента последнего begin(). При следующем begin() счётчик обнуляется. Значение читаем после end(), когда отправлена и последняя накопленная порция.

Число отправок растёт при смене текстуры, заполнении буфера и изменении некоторых настроек. Например, переход к другому режиму смешивания может потребовать отправить уже накопленные изображения.

Атласы из главы 5 помогают уменьшить переключения текстур. Но группировать объекты по текстуре можно только с сохранением нужного перекрытия. Сначала добиваемся правильной картинки, затем измеряем нагрузку. Подробную диагностику нескольких проходов и разных систем отрисовки разберём в разделе 15.1.

Теперь у сцены есть понятный порядок слоёв, а цвет и смешивание меняются только там, где этого требует эффект.

В следующем разделе добавим анимацию персонажа.

4.3. Анимация персонажа

Покадровая анимация — это последовательность изображений, которые сменяют друг друга через заданное время. В LibGDX для неё используется Animation<TextureRegion>: каждый кадр представляет собой область текстуры.

Анимация отвечает за внешний вид. Перемещение, столкновения и нанесение урона остаются задачами игровой логики.

Подготавливаем кадры

Кадры можно вырезать из общей текстуры через TextureRegion.split() или получить из атласа. Используем второй вариант. Атлас объединяет небольшие изображения в одну или несколько текстурных страниц, а файл .atlas хранит имена и границы их областей.

Для примера подготовим четыре группы изображений:

Имена файлов	Анимация
`idle_0.png`, `idle_1.png`, …	Ожидание
`walk_0.png`, `walk_1.png`, …	Ходьба
`attack_0.png`, `attack_1.png`, …	Атака
`death_0.png`, `death_1.png`, …	Смерть

Число после подчёркивания задаёт порядок кадров. Все картинки в нашем примере имеют одинаковые размеры, например 64 × 64 пикселя, и одну точку опоры под ногами. Персонаж не должен менять масштаб или смещаться внутри холста при переходе к следующему кадру.

Собираем атлас для примера

Чтобы не откладывать запуск анимации до следующей главы, подготовим атлас сейчас. Подробно упаковку и организацию ресурсов разберём в разделе 5.1.

В блок dependencies файла lwjgl3/build.gradle добавим инструмент упаковки:

```
implementation "com.badlogicgames.gdx:gdx-tools:$gdxVersion"
```

После обновления Gradle-проекта создадим в настольном модуле отдельный класс PackCharacter. Исходные картинки положим в art/character относительно корня проекта.

```
import com.badlogic.gdx.tools.texturepacker.TexturePacker;

public class PackCharacter {
    public static void main(String[] args) {
        TexturePacker.Settings settings =
            new TexturePacker.Settings();
        settings.useIndexes = true;
        settings.rotation = false;
        settings.stripWhitespaceX = false;
        settings.stripWhitespaceY = false;

        TexturePacker.process( settings, "art/character",
            "assets/atlases",
            "character"
        );
    }
}
```

Запускаем его main() из IDE, установив рабочим каталогом корень проекта. Полученные character.atlas и PNG-страницы находятся в assets/atlases и должны попадать в ресурсы игры. Упаковка выполняется при подготовке графики, а не в игровом цикле.

useIndexes = true сохраняет числовые индексы кадров. Поворот и обрезка прозрачных краёв отключены, поэтому кадры сохраняют исходный прямоугольник — это важно для нашей простой отрисовки без дополнительных смещений.

Загрузим результат в игровом экране, созданном после инициализации приложения:

```
TextureAtlas atlas = new TextureAtlas(
    Gdx.files.internal("atlases/character.atlas")
);
```

Вместе с .atlas нужны все указанные в нём текстурные страницы. Исходная папка art/character для запуска готовой игры не требуется.

Создаём Animation

atlas.findRegions("walk") возвращает области с именем walk в порядке числовых индексов:

```
Array<TextureAtlas.AtlasRegion> frames =
    atlas.findRegions("walk");
if (frames.size == 0) {
    throw new IllegalStateException("Animation not found: walk");
}

Animation<TextureRegion> walk = new Animation<>(
    0.1f, frames, Animation.PlayMode.LOOP
);
```

0.1f — длительность одного кадра в секундах, а не всей анимации. При шести кадрах один цикл длится 0.6 секунды. LOOP повторяет последовательность после её завершения.

Animation, TextureRegion и TextureAtlas находятся в пакете com.badlogic.gdx.graphics.g2d, а Array — в com.badlogic.gdx.utils.

Время анимации

У каждого персонажа есть собственное время текущей анимации:

```
stateTime += delta;
TextureRegion frame = walk.getKeyFrame(stateTime);
```

Время передаём в секундах, как в разделе 3.2. При редких обновлениях промежуточные кадры могут быть пропущены, но анимация не должна замедляться только из-за меньшего числа отрисовок.

Animation выбирает кадр по переданному времени, а не ведёт отдельный таймер каждого персонажа. Одни наборы кадров можно использовать для нескольких объектов, сохраняя их stateTime независимо.

Время увеличиваем на этапе обновления. При отрисовке только получаем и показываем кадр. Если персонаж рисуется в нескольких проходах, повторное увеличение таймера ускорило бы анимацию.

Состояния персонажа

Нам нужны четыре состояния:

Состояние	Воспроизведение
IDLE	Ожидание, повторяется
MOVE	Движение, повторяется
ATTACK	Одна атака, затем возврат к ожиданию или движению
DEAD	Смерть, затем последний кадр сохраняется

Соберём небольшой класс графического представления. Он получает готовый атлас с группами idle, walk, attack и death. Координаты, здоровье и скорость остаются в игровом объекте.

```
import com.badlogic.gdx.graphics.g2d.Animation;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.graphics.g2d.TextureAtlas;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.utils.Array;

import java.util.EnumMap;

public class CharacterAnimation {
    public enum State {
        IDLE, MOVE, ATTACK, DEAD
    }

    private final EnumMap<State,
        Animation<TextureRegion>> animations =
        new EnumMap<>(State.class);
    private State state = State.IDLE;
    private float stateTime;

    public CharacterAnimation(TextureAtlas atlas) {
        animations.put(State.IDLE, create(atlas, "idle", 0.15f, true));
        animations.put(State.MOVE, create(atlas, "walk", 0.1f, true));
        animations.put(State.ATTACK, create(atlas, "attack",
            0.08f, false));
        animations.put(State.DEAD, create(atlas, "death", 0.12f, false));
    }

    private Animation<TextureRegion> create(TextureAtlas atlas,
            String name, float frameDuration, boolean loop) {
        Array<TextureAtlas.AtlasRegion> frames =
            atlas.findRegions(name);
        if (frames.size == 0) {
            throw new IllegalStateException(
                "Animation not found: " + name);
        }

        return new Animation<>( frameDuration, frames,
            loop ? Animation.PlayMode.LOOP : Animation.PlayMode
                .NORMAL
        );
```

CharacterAnimation — продолжение листинга

```
    }

    private void changeState(State next) {
        if (state == next) {
            return;
        }

        state = next;
        stateTime = 0f;
    }

    public void update(float delta, boolean moving,
            boolean attackStarted, boolean dead) {
        if (dead) {
            changeState(State.DEAD);
        } else if (state != State.DEAD) {
            if (attackStarted) {
                state = State.ATTACK;
                stateTime = 0f;
            } else if (state != State.ATTACK) {
                changeState(moving ? State.MOVE : State.IDLE);
            }
        }

        stateTime += delta;

        if (state == State.ATTACK
                && animations.get(state).isAnimationFinished(
                    stateTime)) {
            changeState(moving ? State.MOVE : State.IDLE);
        }
    }

    public void draw(SpriteBatch batch, float x, float y) {
        TextureRegion frame =
            animations.get(state).getKeyFrame(stateTime);
        batch.draw(frame, x - 32f, y, 64f, 64f);
    }
}
```

Размер 64 × 64 выбран для примера. Как в разделе 4.2, x и y задают точку опоры под центром изображения. В другом масштабе игры ширину и высоту следует выразить в единицах её мира.

Смерть имеет наивысший приоритет. Пока играет атака, для изображения выбираются её кадры, но это не запрещает игровому объекту двигаться. После завершения атаки выбираем движение или ожидание по актуальному значению moving.

Подключаем к экрану

Продолжаем устройство GraphicsScreen из раздела 4.1. Для этого примера вместо полей texture и playerRegion создаём атлас и графическое представление персонажа; batch и viewport оставляем:

```
private final TextureAtlas atlas = new TextureAtlas(
    Gdx.files.internal("atlases/character.atlas")
);
private final CharacterAnimation character =
    new CharacterAnimation(atlas);
```

После получения команд и обновления игровой логики вызываем:

```
character.update(delta, moving, attackStarted, dead);
```

moving, attackStarted и dead — данные нашей игры. Они сообщают о фактическом движении, начале новой атаки и смерти. Движение определяется результатом логики, а не только удерживаемой кнопкой: например, стоящему у стены персонажу можно показывать ожидание.

attackStarted передаём как разовое событие начала атаки, которую разрешила логика оружия. На следующем обновлении, если новой атаки нет, значение должно быть false. Удерживание кнопки само по себе не означает, что новая атака начинается каждый кадр.

В открытом проходе batch прежний вызов отрисовки одного изображения заменяем:

```
character.draw(batch, playerX, playerY);
```

playerX и playerY — положение точки опоры персонажа в игровом мире. Таймер здесь не меняется: отрисовка показывает состояние, подготовленное в update().

Атлас принадлежит экрану, поэтому в его dispose() вместо texture.dispose() вызываем atlas.dispose(). CharacterAnimation использует области атласа и не освобождает общую текстуру самостоятельно. При загрузке через AssetManager владельцем будет менеджер, как разберём в

главе 5.

Когда сбрасывать таймер

changeState() обнуляет время только при переходе в другое состояние. Если каждый кадр заново запускать MOVE с нулевым временем, персонаж будет перемещаться, но останется на первом кадре ходьбы.

Новое событие attackStarted имеет другой смысл: оно начинает новую атаку. Поэтому в этой ветке время обнуляется даже тогда, когда текущее состояние уже ATTACK. Так графическое представление не пропускает новую атаку, разрешённую игровыми правилами.

Если оружие должно ждать окончания предыдущего действия, это ограничение проверяет его логика: пока атака не разрешена, новое событие не отправляется.

Режим NORMAL после окончания последовательности возвращает последний кадр. Поэтому анимация смерти не повторяется и не возвращается к первому изображению. В нашем примере DEAD сохраняется до удаления персонажа. Для возрождения или возврата объекта в пул потребуется явный сброс этого состояния.

В разделе 10.2 свяжем анимации с боевой логикой подробнее. Там состояние атаки будет определяться длительностью игрового действия. Текущий класс показывает самостоятельный визуальный цикл с возвратом после заверше-

ния кадров; эти два способа не нужно одновременно применять к одному персонажу без общего правила переходов.

Направление и игровые события

В игре с видом сверху можно поворачивать изображение по углу персонажа, если графика рассчитана на такой способ. Другой вариант — отдельные наборы кадров для направлений. Тогда выбираем нужный набор, сохраняя фазу ходьбы при смене направления, чтобы она не начиналась заново при каждом повороте.

Для отражения влево и вправо не вызываем flip() каждый кадр на общей области атласа: метод меняет саму область, которой могут пользоваться другие персонажи. Используем собственный Sprite или отдельные копии TextureRegion и меняем отражение при смене направления.

Выстрел, звук и урон не должны происходить просто потому, что рисуется картинка атаки. Один кадр анимации может оставаться видимым несколько обновлений, а при низком FPS — вообще не попасть в отрисовку.

Событие выполняем один раз по правилам оружия. Если оно привязано к моменту анимации, проверяем пересечение заданного времени между предыдущим и текущим обновлением, а не только совпадение номера кадра. При новом действии сбрасываем признак уже выполненного события. Практическую связь с боем рассмотрим в разделе 10.2.

В этой главе мы разобрали отрисовку изображения, перекрытие объектов, прозрачность и покадровую анимацию. Теперь у персонажа есть графическое представление, связанное с его состоянием. В следующей главе организуем хранение и загрузку ресурсов, чтобы эти изображения и другие данные можно было использовать во всей игре.

Глава 5. Ресурсы и экран загрузки

В предыдущей главе мы рисовали изображения и собрали анимацию персонажа. Теперь организуем ресурсы игры: подготовим атласы, загрузим общие данные и покажем экран загрузки. Для каждого ресурса определим, кто им пользуется и когда его можно освободить.

План главы

- 5.1. Атласы текстур
- 5.2. Загрузка ресурсов через AssetManager
- 5.3. Экран загрузки
- 5.4. Освобождение ресурсов

5.1. Атласы текстур

В разделе 4.3 мы упаковали кадры персонажа в атлас. Тот же подход подходит для предметов, оружия, кнопок и эффектов: исходные изображения остаются отдельными файлами, а игра использует подготовленные текстурные страницы.

Атлас позволяет разместить несколько изображений на одной текстуре и обращаться к ним по именам. Так изображения, которые рисуются рядом в одном проходе, могут использовать общую текстуру. Как это влияет на отправку данных, мы разобрали в разделе 4.2.

Как устроен атлас

TexturePacker собирает исходные изображения и создаёт:

• одну или несколько текстурных страниц, обычно .png;

• файл .atlas, содержащий имена изображений, их расположение и

дополнительные параметры упаковки.

Например, из player.png, bullet.png и torch.png можно получить game.atlas и game.png. В игровом коде используем имена player, bullet и torch, без расширения файла.

Если все картинки не помещаются на одну страницу, упаковщик создаёт несколько. **Один файл .atlas не обяза-**

тельно означает одну текстуру. Описание ссылается на все страницы, поэтому в ресурсы игры должен попасть весь полученный комплект.

TextureAtlas читает это описание и загружает страницы. Область отдельного изображения представлена TextureAtlas.AtlasRegion, который наследуется от TextureRegion. Помимо границ области, он может хранить индекс кадра и сведения об обрезке исходной картинки.

Настройки упаковки

Утилиту PackCharacter и зависимость gdx-tools мы уже подготовили в разделе 4.3. Полный класс заново писать не нужно. Упаковка по-прежнему выполняется при подготовке ресурсов, а не в игровом цикле.

Для первых примеров сохраняем прежние настройки: rotation = false, stripWhitespaceX = false и stripWhitespaceY = false. Картинки не поворачиваются и не обрезаются, поэтому их можно рисовать обычным batch.draw(region, x, y, width, height).

Поворот и обрезка могут сделать упаковку плотнее, но требуют учитывать дополнительные данные области. Особенно это важно для анимаций: без поправки на обрезанные края персонаж может смещаться при смене кадра. К таким настройкам стоит переходить, когда графический слой умеет их обрабатывать.

При линейной фильтрации есть другая проблема: возле края области в цвет может попасть соседнее изображение. Чтобы уменьшить такую примесь, добавим отступы и дублирование крайних пикселей:

```
settings.paddingX = 2;
settings.paddingY = 2;
settings.duplicatePadding = true;
```

Эти строки размещаем после создания settings, до вызова TexturePacker.process(). Они задают исходную настройку для наших примеров; при уменьшении текстур и использовании mipmap результат дополнительно проверяем на целевых масштабах.

Подготавливаем общий атлас игры

Для следующих разделов используем небольшой атлас game.atlas. Подготовим изображения player.png, bullet.png и torch.png в каталоге art/game.

В main() утилиты из раздела 4.3 после упаковки персонажа добавим ещё один вызов:

```
TexturePacker.process( settings, "art/game", "assets/atlases", "game" );
```

Пути отсчитываются от рабочего каталога утилиты — корня проекта. Результат находится в assets/atlases, чтобы попасть в ресурсы приложения.

Теперь утилита подготавливает два комплекта: character.atlas для анимаций из главы 4 и game.atlas для примера загрузки общих ресурсов. Это разные атласы с разным содержимым. В следующем разделе путь atlases/game.atlas будет указан в GameAssets.

Если изменили исходную картинку, заново запускаем упаковку и используем обновлённые описание и страницы вместе. Замена файла в art/game сама по себе не меняет уже подготовленный атлас.

Получаем изображение по имени

Для знакомства загрузим атлас напрямую. Ниже — поля и метод подготовки игрового экрана. TextureAtlas и TextureRegion находятся в пакете com.badlogic.gdx.graphics.g2d:

```
private TextureAtlas atlas;
private TextureRegion playerRegion;

private void loadAtlas() {
    atlas = new TextureAtlas( Gdx.files.internal("atlases/game.atlas") );

    playerRegion = atlas.findRegion("player");
    if (playerRegion == null) {
        throw new IllegalStateException( "Region not found: player");
    }
}
```

loadAtlas() вызываем один раз при подготовке экрана после инициализации приложения. К моменту отрисовки атлас и playerRegion уже должны быть готовы.

Имена и регистр должны совпадать с результатом упаковки. Проверка на null позволяет сразу увидеть, какой области не хватает, вместо ошибки при первой попытке её нарисовать.

Найденную область используем как любую другую TextureRegion. Например, при настроенной камере и уже созданном batch:

```
batch.begin();
batch.draw(playerRegion, 208f, 103f, 64f, 64f);
batch.end();
```

Область сохраняем в поле, а не ищем по имени каждый кадр. Она ссылается на текстурную страницу и не создаёт отдельную копию пикселей.

Для одиночного изображения используем findRegion(). Для последовательности кадров с одним именем и числовыми индексами — findRegions(), как в анимации из раздела 4.3. Индексы задаются суффиксами _0, _1 и далее при включённом useIndexes.

Как разделять ресурсы

Атласы удобно группировать по тому, какие изображения загружаются и используются вместе:

Группа	Пример содержимого
Общий интерфейс	Кнопки, панели, значки
Персонажи и эффекты	Кадры анимаций, снаряды, вспышки
Отдельная локация	Её окружение и декорации

Это отправная точка, а не обязательное деление для каждой игры. Если ресурсы всегда нужны одновременно, их можно объединить. Если уровни имеют много разной графики, отдельные комплекты позволяют загружать только нужное окружение.

При загрузке TextureAtlas загружаются все его страницы, даже если экран использует одну область. Поэтому объединять всю игру в огромный атлас необязательно: небольшому меню тогда придётся держать в памяти графику других сцен.

Учитываем также фильтрацию. Она задаётся для текстурной страницы, а не независимо для каждой области. Если части интерфейса нужна линейная фильтрация, а пиксельным изображениям — Nearest, их размещение на одной странице усложнит настройку. Часто удобнее разделить такие ресурсы.

Атлас уменьшает переключения текстур, но не гарантирует один вызов отрисовки. Несколько страниц, настройки batch и нужный порядок перекрытия продолжают влиять на результат. Ради объединения изображений нельзя нарушать порядок слоёв из раздела 4.2.

Кто освобождает атлас

В показанном варианте экран загрузил атлас напрямую и владеет им. Когда экран окончательно перестаёт использовать его области, освобождаем атлас:

```
atlas.dispose();
```

Это освобождает все принадлежащие атласу текстурные страницы. Отдельно вызывать dispose() для playerRegion или уничтожать страницу через playerRegion.getTexture().dispose() не нужно: на ту же текстуру могут ссылаться другие области.

После освобождения атласа сохранённые области больше нельзя использовать для отрисовки. Если атлас общий для нескольких экранов, его время жизни должен контролировать общий владелец. Подробный порядок освобождения разберём в разделе 5.4.

Теперь у нас есть подготовленные атласы и доступ к изображениям по именам. В следующем разделе передадим загрузку общих ресурсов AssetManager, чтобы экраны получали готовые объекты из одного места.

5.2. Загрузка ресурсов через AssetManager

В предыдущем разделе экран загружал атлас напрямую. Если один атлас нужен меню и игровому экрану, создавать отдельную копию для каждого неудобно: текстуры дублируются, а время их жизни приходится согласовывать вручную.

AssetManager хранит загруженные ресурсы, учитывает зависимости и управляет их временем жизни. Создадим один менеджер на экземпляр игры и передадим доступ к нему нужным экранам. При переключении экранов менеджер продолжает работать; создавать его заново для каждого перехода не нужно.

Подготавливаем очередь

Используем game.atlas из раздела 5.1. Дополнительно поместим в ресурсы приложения короткий звук выстрела audio/shot.ogg и музыкальную запись audio/music.ogg. В проекте с каталогом assets они находятся в assets/audio.

Соберём пути и очередь первоначальной загрузки в одном классе:

```
import com.badlogic.gdx.assets.AssetManager;
import com.badlogic.gdx.audio.Music;
import com.badlogic.gdx.audio.Sound;
import com.badlogic.gdx.graphics.g2d.TextureAtlas;

public final class GameAssets {
    public static final String ATLAS = "atlases/game.atlas";
    public static final String SHOT = "audio/shot.ogg";
    public static final String MUSIC = "audio/music.ogg";

    private GameAssets() {
    }

    public static void queue(AssetManager assets) {
```

GameAssets — продолжение листинга

```
        assets.load(ATLAS, TextureAtlas.class);
        assets.load(SHOT, Sound.class);
        assets.load(MUSIC, Music.class);
    }
}
```

Второй аргумент load() указывает тип ресурса и позволяет менеджеру выбрать загрузчик. Для TextureAtlas, Sound и Music стандартные загрузчики уже предусмотрены. Sound используем для коротких эффектов, Music — для длинных записей; воспроизведение разберём в разделе 13.3.

Пути отсчитываются от каталога ресурсов, как при Gdx.files.internal(). Имена и регистр должны совпадать с

файлами. Вместе с .atlas сохраняем все его текстурные страницы.

Менеджер создаём один раз в create() главного класса. Если assets — поле этого класса типа AssetManager, подготовка выглядит так:

```
assets = new AssetManager();
GameAssets.queue(assets);
```

Полный главный класс и переход между экранами соберём в разделе 5.3. Здесь GameAssets задаёт список ресурсов, а приложение владеет менеджером.

Обрабатываем загрузку по кадрам

load() добавляет задание в очередь. Он не означает, что ресурс уже доступен. Для продвижения загрузки вызываем update() из игрового цикла:

```
boolean finished = assets.update();
float progress = assets.getProgress();
```

Пока finished равен false, продолжаем показывать экран загрузки. Когда очередь завершена, update() возвращает true

и можно перейти к использованию подготовленных ресурсов.

getProgress() возвращает значение от 0 до 1. Для процентов умножаем его на 100, для полосы с диапазоном 0–1 используем напрямую. Это состояние очереди, а не точная доля прочитанных байтов или оценка оставшегося времени. Решение о завершении принимаем по результату update(), а не по округлённой надписи 100%.

Для нашего экрана загрузки достаточно одного вызова update() за кадр, после которого обновляем индикатор и рисуем интерфейс. Метод продолжает работу очереди между кадрами; одного getProgress() для загрузки недостаточно.

Загрузчики могут выполнять часть работы в фоне, но создание графических ресурсов требует потока с OpenGL-контекстом. Поэтому не переносим assets.update() в произвольный рабочий поток. Обработка отдельного тяжёлого ресурса всё ещё может задержать кадр: асинхронная очередь не гарантирует полного отсутствия пауз.

Если файл отсутствует или загрузка завершается ошибкой, стандартный менеджер сообщает об этом исключением. В следующем разделе обработаем этот случай на экране загрузки, чтобы игрок видел сообщение вместо бесконечного ожидания.

Получаем готовые ресурсы

После завершения очереди получаем нужные объекты:

```
TextureAtlas atlas = assets.get( GameAssets.ATLAS, TextureAtlas.class );
Sound shot = assets.get(GameAssets.SHOT, Sound.class);
Music music = assets.get(GameAssets.MUSIC, Music.class);
```

get() возвращает уже загруженный объект. Он не запускает загрузку, не ждёт её завершения и не создаёт новую копию. Если тому же менеджеру принадлежит общий атлас, другой экран получает его через такой же get(), без повторного load().

Из полученного атласа выбираем области по правилам раздела 5.1. Например, на этапе подготовки экрана:

```
TextureRegion playerRegion = atlas.findRegion("player");
if (playerRegion == null) {
    throw new IllegalStateException("Region not found: player");
}
```

Область сохраняем для последующей отрисовки. Успешная загрузка самого атласа не гарантирует, что в нём присутствует каждое имя, ожидаемое игровым кодом.

Если один ресурс нужен до завершения всей очереди, проверяем его отдельно:

```
if (assets.isLoaded(GameAssets.ATLAS, TextureAtlas.class)) {
    TextureAtlas atlas = assets.get( GameAssets.ATLAS, TextureAtlas.class
    );
}
```

Это альтернативный способ получить ресурс по мере готовности. Для обычного перехода в игру в нашем примере ждём завершения всей первоначальной очереди.

Текстурные страницы атласа загружаются как его зависимости. Отдельно ставить каждую PNG-страницу в очередь ради использования этого атласа не нужно.

Не превращаем загрузку в блокирующее ожидание

finishLoading() ждёт завершения очереди. Это удобно в небольшом тесте, где загрузка должна закончиться до продолжения работы. Для обновляемого экрана загрузки используем другой порядок: шаг очереди, обновление индикатора, отрисовка кадра.

Цикл while (!assets.update()) внутри render() не позволяет закончить кадр, пока очередь не завершится. Игрок увидит неподвижный экран вместо обновляющегося прогресса.

Первоначальную очередь формируем один раз. Новые задания, добавленные во время загрузки, могут изменить показанный прогресс. Повторный load() уже известного ресурса учитывается в числе ссылок, которое позже нужно согла-

совать с unload(). Поэтому не вызываем GameAssets.queue() каждый кадр или при каждом открытии экрана.

Полученные из менеджера атлас, звуки и музыку экран самостоятельно не освобождает. Их временем жизни управляет приложение через менеджер; правила выгрузки разберём в разделе 5.4.

Теперь ресурсы подготавливаются общей очередью, а экраны получают готовые объекты из одного места. В следующем разделе соберём экран загрузки с фоном, надписью, полосой прогресса и переходом в игру.

5.3. Экран загрузки

В моей игре экран загрузки показывает фоновое изображение, надпись и полосу прогресса. Для интерфейса используются Label и ProgressBar из Scene2D.

Рассмотрим тот же подход в самостоятельном примере. Экран будет обновлять очередь AssetManager один раз за кадр, показывать результат и переходить в игру после завершения загрузки.

Минимум Scene2D для этого примера

Scene2D позволяет собирать интерфейс из элементов, которые сами отвечают за свою отрисовку. Нам понадобятся пять классов:

• Stage — сцена, которая хранит элементы интерфейса, обновляет и

рисует их.

• Table — контейнер, размещающий элементы по строкам и ячейкам.

• Label — текстовая надпись.

• ProgressBar — полоса, отображающая значение в заданном диапазоне.

• Skin — набор ресурсов и стилей оформления: шрифтов, изображений,

цветов и настроек элементов.

В каждом кадре вызываем stage.act(delta) для обновления элементов и их действий, затем stage.draw() для отрисовки. Размещение надписи и полосы поручим таблице. Полное устройство интерфейса разберём в разделе 14.1; здесь достаточно этой схемы.

Ресурсы самого экрана

Экран загрузки должен работать до появления основных ресурсов. Поэтому небольшой набор для него подготавливаем отдельно:

- фоновую картинку assets/images/logo.jpg;
- файлы оформления в assets/ui: uiskin.json, uiskin.atlas, uiskin.png

и default.fnt.

Для первого запуска можно взять комплект из тестов LibGDX, на который ссылается документация Skin[6]. Копируем все четыре файла вместе. Фоновое изображение выбираем своё. В коде пути начинаются с images/ и ui/, без префикса assets/.

В выбранном skin нужны стиль Label с именем default и стиль горизонтального ProgressBar с именем default-horizontal. Для русских надписей шрифт должен содержать кириллицу; пока используем Loading.... Подготовку шрифтов разберём в главе 6.

Эти небольшие ресурсы загрузим напрямую в конструкторе экрана. Основные игровые ресурсы останутся в очереди AssetManager. Так экран не зависит от атласа, загрузку которого он показывает.

Запуск игры

Используем GameAssets из раздела 5.2. Главный класс создаёт общий менеджер и SpriteBatch, затем открывает экран загрузки:

```
import com.badlogic.gdx.Game;
```

[6] документация Skin: https://libgdx.com/wiki/graphics/2d/scene2d/skin

DemoGame — продолжение листинга

```
import com.badlogic.gdx.Screen;
import com.badlogic.gdx.assets.AssetManager;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;

public class DemoGame extends Game {
    public AssetManager assets;
    public SpriteBatch batch;

    @Override
    public void create() {
        assets = new AssetManager();
        batch = new SpriteBatch();
        GameAssets.queue(assets);
        setScreen(new LoadingScreen(this));
    }

    @Override
    public void dispose() {
        Screen current = getScreen();
        setScreen(null);
        if (current != null) {
            current.dispose();
        }
        batch.dispose();
        assets.dispose();
    }
}
```

Один менеджер сохраняется при переключении экранов. Поля открыты для компактности примера. Каждый публичный класс размещаем в отдельном Java-файле; DemoGame,

LoadingScreen, GameScreen и GameAssets — в одном пакете модуля core. Строку package в листингах не показываем: она зависит от имени вашего проекта.

Это самостоятельный вариант приложения: в настольном launcher передаём new DemoGame() вместо new BookGame(). Одноимённый GameScreen из главы 3 заменяем новым или размещаем примеры в разных пакетах.

Фон и интерфейс

Для интерфейса используем ExtendViewport с базовым размером 480 × 270: он сохраняет масштаб по осям одинаковым и при других пропорциях окна расширяет видимую область. Подробно варианты viewport сравним в главе 7.

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.ScreenAdapter;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.Sprite;
import com.badlogic.gdx.scenes.scene2d.Stage;
import com.badlogic.gdx.scenes.scene2d.ui.Label;
```

LoadingScreen — продолжение листинга

```java
import com.badlogic.gdx.scenes.scene2d.ui.ProgressBar;
import com.badlogic.gdx.scenes.scene2d.ui.Skin;
import com.badlogic.gdx.scenes.scene2d.ui.Table;
import com.badlogic.gdx.utils.GdxRuntimeException;
import com.badlogic.gdx.utils.viewport.ExtendViewport;

public class LoadingScreen extends ScreenAdapter {
    private final DemoGame game;
    private final Stage stage;
    private final Skin skin;
    private final Texture backgroundTexture;
    private final Sprite background;
    private final Label loading;
    private final ProgressBar progressBar;
    private boolean failed;

    public LoadingScreen(DemoGame game) {
        this.game = game;
        stage = new Stage(new ExtendViewport(480f, 270f), game.batch);
        skin = new Skin(Gdx.files.internal("ui/uiskin.json"));
        backgroundTexture = new Texture(
            Gdx.files.internal("images/logo.jpg")
        );
        background = new Sprite(backgroundTexture);

        loading = new Label("Loading...", skin);
        progressBar = new ProgressBar(0f, 100f, 1f, false, skin);

        Table layout = new Table();
        layout.setFillParent(true);
        layout.bottom().pad(32f);
        layout.add(loading).center();
        layout.row();
        layout.add(progressBar).growX().height(18f).padTop(12f);
        stage.addActor(layout);
    }

    @Override
    public void render(float delta) {
        if (Gdx.graphics.getWidth() <= 0
            || Gdx.graphics.getHeight() <= 0) {
            return;
        }

        boolean ready = false;
        if (!failed) {
            try {
                ready = game.assets.update();
            } catch (GdxRuntimeException error) {
```

LoadingScreen — продолжение листинга

```
        int progress = (int) (game.assets.getProgress() * 100f);
        progressBar.setValue(progress);
        loading.setText(
            failed ? "Loading failed" : "Loading... " + progress
                + "%"
        );

        Gdx.gl.glClearColor(0f, 0f, 0f, 1f);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        stage.getViewport().apply();

        game.batch.setProjectionMatrix( stage.getCamera().combined);
        game.batch.begin();
        background.draw(game.batch);
        game.batch.end();

        stage.act(delta);
        stage.draw();

        if (ready && !failed) {
            game.setScreen(new GameScreen(game));
            dispose();
            return;
        }
    }

    @Override
    public void resize(int width, int height) {
        if (width <= 0 || height <= 0) return;
        stage.getViewport().update(width, height, true);
        float viewWidth = stage.getViewport().getWorldWidth();
        float viewHeight = stage.getViewport().getWorldHeight();
        float scale = Math.max( viewWidth / backgroundTexture.getWidth(),
            viewHeight / backgroundTexture.getHeight()
        );
        float bgWidth = backgroundTexture.getWidth() * scale;
        float bgHeight = backgroundTexture.getHeight() * scale;
        background.setBounds( (viewWidth - bgWidth) / 2f,
            (viewHeight - bgHeight) / 2f,
            bgWidth, bgHeight
        );
    }

    @Override
    public void dispose() {
        stage.dispose();
        skin.dispose();
        backgroundTexture.dispose();
    }
}
```

setFillParent(true) растягивает корневую таблицу на всю сцену, bottom().pad(32f) размещает содержимое снизу с отступами, а row() начинает строку для полосы прогресса. growX() позволяет полосе занять доступную ширину. Размеры задаются в единицах viewport; таблица сама пересчитывает размещение при изменении окна или надписи.

В конструкторе ProgressBar задаём диапазон 0–100, шаг 1 и горизонтальную ориентацию — аргумент false. Кнопок на этом экране нет, поэтому подключать Stage как обработчик ввода пока не требуется.

Фон масштабируется с сохранением пропорций и полностью закрывает видимую область. При несовпадении пропорций часть изображения выходит за края экрана. Это подходящий вариант для фоновой картинки, но важный логотип лучше размещать отдельным элементом, чтобы он не обрезался.

Фон рисуется первым, затем stage.draw() выводит интерфейс поверх него. Этот метод сам открывает и закрывает batch, поэтому вызываем его после game.batch.end().

Stage использует общий game.batch. Поскольку batch передан извне, stage.dispose() его не уничтожает: за него отвечает DemoGame.

Готовность и ошибка загрузки

Используем порядок из раздела 5.2: один вызов assets.update() за кадр, обновление индикатора, отрисовка. Значение ready определяет переход, а целый процент нужен только для надписи и полосы.

Если update() выбрасывает исключение, устанавливаем failed и прекращаем дальнейшие попытки загрузки. Интерфейс продолжает рисоваться с надписью Loading failed, а причина ошибки записывается в лог. В этом примере повторной попытки нет: сначала исправляем ресурс и перезапускаем приложение.

Эта обработка относится к основной очереди. Ошибка в самом skin или фоновой картинке возникнет раньше, при создании экрана, поэтому его небольшой комплект ресурсов нужно проверить отдельно.

Переход в игру

Для проверки создадим минимальный экран, который использует уже загруженный атлас:

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.ScreenAdapter;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.g2d.TextureAtlas;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
```

GameScreen — продолжение листинга

```java
import com.badlogic.gdx.utils.GdxRuntimeException;
import com.badlogic.gdx.utils.viewport.FitViewport;

public class GameScreen extends ScreenAdapter {
    private final DemoGame game;
    private final FitViewport viewport = new FitViewport(480f, 270f);
    private final TextureRegion player;

    public GameScreen(DemoGame game) {
        this.game = game;
        TextureAtlas atlas = game.assets.get(
            GameAssets.ATLAS, TextureAtlas.class
        );
        player = atlas.findRegion("player");
        if (player == null) {
            throw new GdxRuntimeException( "Region not found: player");
        }
    }

    @Override
    public void render(float delta) {
        if (Gdx.graphics.getWidth() <= 0
            || Gdx.graphics.getHeight() <= 0) {
            return;
        }

        Gdx.gl.glClearColor(0.12f, 0.14f, 0.16f, 1f);
        Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
        viewport.apply();
        game.batch.setProjectionMatrix( viewport.getCamera().combined);
        game.batch.begin();
        game.batch.draw(player, 208f, 103f, 64f, 64f);
        game.batch.end();
    }

    @Override
    public void resize(int width, int height) {
        if (width <= 0 || height <= 0) return;
        viewport.update(width, height, true);
    }
}
```

После запуска увидим фон с индикатором, затем персонажа на тёмном игровом экране. Если ресурсов мало, загру-

зочный экран может промелькнуть: искусственно задерживать переход не нужно.

Если после загрузки файлов проекту нужна дополнительная подготовка ресурсов или мира, переход в игру разрешаем только после её завершения. Такую подготовку тоже выполняем по шагам, без блокирующего ожидания в render().

Game.setScreen() не уничтожает предыдущий экран автоматически. Наш LoadingScreen одноразовый, поэтому после переключения освобождаем его stage, skin и фоновую текстуру. Атлас, звуки, музыка и общий batch продолжают использоваться игрой.

Здесь skin принадлежит только загрузочному экрану. Общий skin, используемый другими экранами или загруженный через AssetManager, так освобождать нельзя. Владение и порядок освобождения ресурсов разберём в следующем разделе.

5.4. Освобождение ресурсов

После загрузки нужно решить ещё две задачи: кто владеет ресурсом и когда его можно освободить. Например, игровой экран хранит область персонажа, но это не означает, что ему принадлежит весь атлас.

Сборщик мусора освобождает обычные Java-объекты. У текстур, звуков и других ресурсов LibGDX есть также память и объекты вне Java-кучи, поэтому для них требуется явное освобождение через dispose(). Присваивание null полю с текстурой не заменяет этот вызов.

Владелец и пользователь ресурса

Владелец определяет время жизни ресурса. Остальные объекты могут использовать его, но не должны уничтожать по своему усмотрению.

В примере из раздела 5.3 обязанности распределены так:

Ресурс	Кто управляет его временем жизни	Когда освобождаем
Атлас, звук и музыка из очереди	`DemoGame` через `AssetManager`	Через `unload()` при ненадобности или `assets.dispose()` при завершении игры
Общий `SpriteBatch`	`DemoGame`	При завершении игры, после экранов
`Stage`, собственный `Skin` и фоновая текстура	`LoadingScreen`	После перехода в игру или при завершении приложения на экране загрузки
Область `TextureRegion`	Отдельным ресурсом не владеет	Собственного `dispose()` нет; время её использования ограничено временем жизни текстуры

Кто освобождает ресурсы

Раздел 6.4 · владение ресурсами в примере главы 6

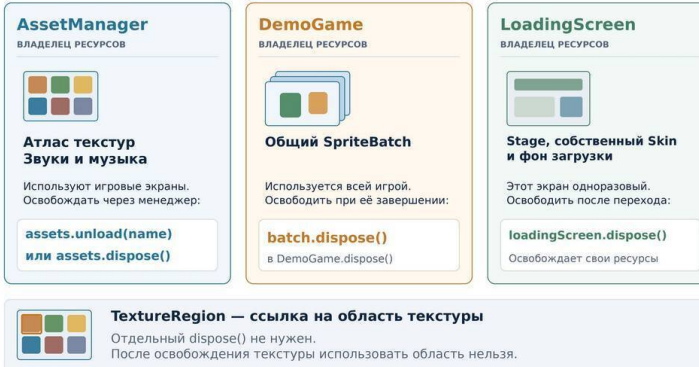




Рисунок 5.1. Владельцы ресурсов в примере главы 5 и порядок освобождения при

завершении игры.

GameScreen получает атлас через менеджер и сохраняет область player. Он пользуется общими ресурсами, но не освобождает ни атлас, ни game.batch. Поэтому в его минимальном варианте отдельный dispose() не понадобился. Если экран создаст собственный ресурс, ответственность за него

нужно добавить явно.

Ресурсы из AssetManager

Ресурс, полученный из менеджера, вручную не уничтожаем. Например, такой вызов ошибочен:

```
assets.get(GameAssets.ATLAS, TextureAtlas.class).dispose();
```

Менеджер продолжит хранить объект, чьи текстурные страницы уже освобождены. Другой экран может получить тот же атлас через get() и попытаться рисовать его области.

Если ресурс больше не нужен, освобождаем его через менеджер:

```
assets.unload(GameAssets.ATLAS);
```

Этот фрагмент относится к моменту, когда все пользователи атласа закончили работу. Он не должен выполняться после каждого перехода между экранами. При фактической выгрузке менеджер освобождает ресурс и учитывает его зависимости — в случае атласа это текстурные страницы. Страницы отдельно не выгружаем, если не ставили их в очередь как самостоятельные ресурсы.

При завершении игры достаточно вызвать assets.dispose():

менеджер освободит оставшиеся ресурсы и завершит свою работу. Перечислять перед этим все атласы, звуки и музыку отдельными вызовами unload() не требуется. После dispose() этот менеджер больше не используем.

Что считает менеджер

AssetManager ведёт счётчики ссылок на ресурсы. Повторная обработанная загрузка увеличивает счётчик, а unload() уменьшает его. Пока счётчик остаётся положительным, ресурс сохраняется. Зависимости также учитываются: общая текстурная страница может ещё требоваться другому загруженному ресурсу.

Вызов get() не увеличивает счётчик. Сохранённая в поле Java-ссылка тоже не сообщает менеджеру, что ресурс ещё нужен.

Предположим, приложение один раз загрузило game.atlas, а меню и игровой экран получили его через get(). Это не три удерживающие ссылки менеджера. Если приложение выгрузит атлас, обе сохранённые ссылки станут непригодны для отрисовки, хотя сами Java-объекты продолжат существовать.

Поэтому счётчик не заменяет правила владения. В нашей схеме общие ресурсы загружает DemoGame, экраны получают готовые объекты, а решение о выгрузке остаётся у приложения. Повторять load() при каждом открытии экрана ради получения ресурса не нужно.

Общие ресурсы и ресурсы локации

Общий интерфейс можно держать загруженным всё время, а окружение отдельной локации — пока она используется. При смене карты сначала прекращаем работу её объектов и рендереров, затем выгружаем ненужные ресурсы и подготавливаем следующую локацию.

После выгрузки старые объекты не должны продолжать рисовать её текстуры. Это касается и объектов, которые сохранились в списке или ожидают повторного использования: наличие TextureRegion в поле не продлевает жизнь текстуры.

В небольшом проекте проще один раз загрузить общие ресурсы и держать их до выхода. Поэтапную выгрузку вводим, когда её требует объём памяти. В примере этой главы game.atlas, звук и музыка остаются загруженными до завершения игры.

Экран и ресурс — разные времена жизни

hide() означает, что экран перестал быть текущим. Он может позже вернуться, поэтому общий атлас автоматически выгружать в этом методе не стоит. Жизненный цикл экранов мы разобрали в разделе 3.3.

Наш LoadingScreen одноразовый: после перехода в игру

его собственные ресурсы больше не нужны. Поэтому в конце его render() после setScreen() вызывается dispose(). Если экран сохраняем для повторного использования, освобождаем его только при окончательном удалении.

У Stage и оформления также могут быть разные владельцы. В разделе 5.3 сцена получила готовый game.batch, поэтому stage.dispose() его не уничтожает. При создании new Stage(viewport) без передачи batch сцена создаёт собственный batch и освобождает его сама.

stage.dispose() не освобождает наш Skin и фоновую текстуру. Они созданы отдельно и принадлежат загрузочному экрану, поэтому в LoadingScreen.dispose() есть отдельные вызовы. Если skin общий для нескольких экранов, его освобождение передаём общему владельцу.

При завершении приложения соблюдаем порядок из DemoGame.dispose(): 1. Снимаем текущий экран и освобождаем его собственные ресурсы. 2. Освобождаем общий SpriteBatch. 3. Освобождаем AssetManager с оставшимися ресурсами.

Если приложение хранит несколько экранов для возврата, при завершении нужно освободить и их. В нашем примере сохраняется только текущий экран, а загрузочный уже уничтожен после перехода.

Ресурсы, загруженные напрямую

Для ресурса, созданного без менеджера, ответственность остаётся у его владельца:

```
TextureAtlas atlas = new TextureAtlas(
    Gdx.files.internal("atlases/game.atlas")
);

// Когда все пользователи атласа закончили работу:
atlas.dispose();
```

Это альтернативный способ загрузки, как в разделе 5.1. Он не нужен дополнительно к загрузке того же атласа через менеджер: иначе получим две копии ресурса с разными владельцами.

При освобождении такого TextureAtlas уничтожаются принадлежащие ему текстурные страницы. Отдельно вызывать region.getTexture().dispose() нельзя: на страницу могут ссылаться другие области. Сами TextureRegion не освобождают текстуру и не содержат собственных копий пикселей.

Для каждого ресурса определяем владельца при создании или загрузке. Тогда смена экрана, выгрузка локации и завершение игры имеют понятный порядок: сначала заканчиваем использование, затем освобождаем ресурс.

Глава 6. Шрифты в LibGDX

План главы

- 6.1. TrueType и BitmapFont
- 6.2. Свой FontGenerator
- 6.3. Размеры и качество текста
- 6.4. Шрифты в браузере, текстурные страницы и память

6.1. TrueType и BitmapFont

Название предмета, здоровье персонажа, текст задания и кнопки меню — всё это требует шрифта. Пока в игре несколько английских надписей, хватает стандартного new BitmapFont(). Но для русского интерфейса, собственного оформления и разных размеров текста нужно подготовить свои шрифты.

В LibGDX можно загрузить готовый растровый шрифт или создать его из файла TrueType. В обоих случаях текст рисует BitmapFont. Разница заключается в том, когда и каким способом мы подготовили изображения символов.

Если нужны несколько заранее известных размеров и одинаковые ресурсы для всех платформ, удобно экспортировать шрифты до запуска игры. Генерация из .ttf позволяет подготовить нужный размер во время загрузки приложения, но требует расширения FreeType. Рассмотрим оба способа.

Что именно мы загружаем

TrueType-файл .ttf содержит описание шрифта, включая контуры символов. Его нельзя передать в конструктор BitmapFont как готовую текстурную страницу. Сначала нужные символы растеризуются — превращаются в изображения выбранного размера.

Растровый шрифт состоит из изображений символов и данных об их расположении. Отдельное изображение символа называется **глифом**. Помимо его размеров, нужны смещения, расстояние до следующего символа и другие параметры размещения текста.

Файл	Что содержит
`ui.ttf`	Исходный TrueType-шрифт для генерации
`ui-32.fnt`	Описание глифов, метрики и ссылки на текстурные страницы
`ui-32.png`	Изображения глифов; страниц может быть несколько

Во время отрисовки BitmapFont использует эти изображения через batch. Он не строит заново контуры букв при каждом вызове draw().

Загружаем готовый растровый шрифт

Если шрифт уже подготовлен, размещаем .fnt и все его текстурные страницы в ресурсах приложения. Пути к страницам должны соответствовать указанным внутри .fnt. Для обычного экспорта с именами файлов без дополнительных каталогов достаточно положить страницы рядом с ним.

Для этого варианта нужны уже экспортированные файлы ui-32.fnt и его страниц. Экспорт через Hiero разберём в разделе 6.4. В игровом экране объявим поле и метод подготов-

ки:

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.g2d.BitmapFont;

// Поле и метод игрового экрана:
private BitmapFont font;

private void initFont() {
    font = new BitmapFont( Gdx.files.internal("fonts/ui-32.fnt") );
}
```

Этот конструктор читает описание и загружает текстурные страницы. Отдельно создавать Texture для каждой страницы не требуется. initFont() вызываем один раз при подготовке экрана после инициализации приложения, до первой отрисовки.

Ниже предполагается, что batch уже создан и ему задана матрица камеры интерфейса. Открываем проход и рисуем надпись; если проход уже открыт, добавляем в него только вызов font.draw():

```
batch.begin();
font.draw(batch, "Здоровье: 100", 24f, 240f);
batch.end();
```

Координаты относятся к системе, которую задаёт матрица batch. Здесь предполагается экранная область интерфейса размером 480 × 270 единиц. Для текста поверх игры исполь-

зуем камеру интерфейса, чтобы движение и зум мировой камеры не меняли положение надписи. Камеру и viewport подробно разберём в главе 7.

Шрифт сохраняем в поле и используем повторно. Создавать его каждый кадр в render() нельзя: это повторная загрузка изображений и создание графических ресурсов.

Создаём BitmapFont из TrueType

Для генерации во время работы приложения используем расширение **gdx- freetype** и класс FreeTypeFontGenerator. При создании проекта его можно выбрать в GDX-Liftoff. Если расширение уже подключено генератором, зависимости повторно не добавляем.

Для существующего проекта в блок dependencies файла core/build.gradle добавляем:

```
api "com.badlogicgames.gdx:gdx-freetype:$gdxVersion"
```

В lwjgl3/build.gradle добавляем нативную библиотеку для настольного запуска:

```
implementation("com.badlogicgames.gdx:" +
    "gdx-freetype-platform:$gdxVersion:natives-desktop")
```

Для Android добавляем нативные библиотеки в блок dependencies файла android/build.gradle. В проекте, созданном GDX-Liftoff, конфигурация natives и копирование этих библиотек уже предусмотрены. Если сборка поддерживает все четыре архитектуры:

```
natives("com.badlogicgames.gdx:" +
    "gdx-freetype-platform:$gdxVersion:natives-arm64-v8a")
natives("com.badlogicgames.gdx:" +
    "gdx-freetype-platform:$gdxVersion:natives-armeabi-v7a")
natives("com.badlogicgames.gdx:" +
    "gdx-freetype-platform:$gdxVersion:natives-x86")
natives("com.badlogicgames.gdx:" +
    "gdx-freetype-platform:$gdxVersion:natives-x86_64")
```

Используем ту же переменную gdxVersion, что и для остальных компонентов LibGDX. После изменения зависимостей обновляем Gradle-проект в IDE.

Следующий пример генерации рассчитан на настольное приложение и Android. Для HTML-сборки на GWT стандартное нативное расширение не подходит. Готовые .fnt и PNG работают и в браузере; общий способ подготовки ресурсов для нескольких платформ разберём в разделе 6.4.

Для второго варианта поместим в assets/fonts файл ui.ttf, содержащий латиницу и кириллицу. В коде используем путь fonts/ui.ttf. Помимо Gdx и BitmapFont, понадобятся импорты:

```
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.Texture.TextureFilter;
import com.badlogic.gdx.graphics.g2d.freetype
    .FreeTypeFontGenerator;
import com.badlogic.gdx.graphics.g2d.freetype
    .FreeTypeFontGenerator.FreeTypeFontParameter;
```

Метод создания шрифта разместим в том же классе, который подготавливает ресурсы:

```
private BitmapFont createFont(int size) {
    FreeTypeFontGenerator generator =
        new FreeTypeFontGenerator( Gdx.files.internal("fonts/ui.ttf") );

    try {
        FreeTypeFontParameter parameter =
            new FreeTypeFontParameter();

        parameter.size = size;
        parameter.color = Color.WHITE;
        parameter.incremental = false;
        parameter.characters =
            FreeTypeFontGenerator.DEFAULT_CHARS
            + "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
            + "абвгдеёжзийклмнопрстуфхцчшщъыьэюя"
            + " —…№";
        parameter.minFilter = TextureFilter.Linear;
        parameter.magFilter = TextureFilter.Linear;

        return generator.generateFont(parameter);
    } finally {
        generator.dispose();
    }
}
```

Если выбираем генерацию вместо загрузки .fnt, заменяем предыдущий initFont():

```
private void initFont() {
    font = createFont(32);
}
```

Это два альтернативных способа подготовки одного поля font. Не вызываем оба подряд: иначе ссылка на первый шрифт потеряется до его освобождения.

Глифы генерируем белыми на прозрачном фоне. Тогда цвет одноцветной надписи можно задавать при отрисовке через font.setColor(). Это правило относится и к экспорту готовых шрифтов; подробно разберём его в разделе 6.2.

parameter.size задаёт размер растеризации в пикселях. Он не гарантирует, что высота каждой буквы будет ровно 32: результат зависит от метрик исходного шрифта. Размер на экране также зависит от масштаба камеры интерфейса; это разберём в разделе 6.3.

В нашем примере incremental = false: нужные глифы создаются сразу. После генерации можно освободить FreeTypeFontGenerator, а полученный BitmapFont продолжит работать. При инкрементальной генерации новые символы добавляются по мере использования, поэтому генератор должен оставаться доступным; для первого примера такой режим не нужен.

Генерация создаёт текстуры, поэтому выполняем её в потоке приложения с графическим контекстом, например при подготовке экрана. Это работа загрузки, а не операция для каждого кадра.

Почему отдельно задаём символы

Наличие кириллицы в .ttf ещё не означает, что она появится в готовом растровом шрифте. Генератор должен получить нужные символы через parameter.characters. Для готового .fnt их выбирают при экспорте.

В примере к стандартному набору добавлены русские буквы, включая Ё и ё, а также тире, многоточие и знак номера. Если интерфейс использует другие языки или специальные знаки, дополняем набор ими. Исходный .ttf тоже должен содержать соответствующие глифы: список символов не создаёт отсутствующий рисунок буквы.

При проверке полезно вывести настоящую строку интерфейса с цифрами и пунктуацией, например: Задание № 3 — найти ещё 10 монет…. Так легче заметить пропущенные символы, чем на английском Hello.

Не включаем весь Unicode просто на всякий случай. Чем больше глифов и чем крупнее их размер, тем больше времени и текстурной памяти требуется для подготовки.

Генерация в памяти и экспорт — разные операции

generateFont() возвращает готовый объект BitmapFont с растровыми глифами в памяти приложения. Он **не сохраняет автоматически** файлы .fnt и PNG в каталог ресурсов.

Если нужны файлы для поставки с игрой, шрифт предварительно экспортируют инструментом вроде Hiero. Тогда приложение только загружает подготовленный результат. Для нескольких фиксированных размеров интерфейса и браузерной сборки такой подход часто удобнее; его разберём отдельно.

Текстурные страницы шрифта похожи на атлас: на них упакованы изображения глифов. Независимая генерация без общего упаковщика обычно создаёт собственные страницы; при большом наборе символов их может быть несколько. Несколько шрифтов также могут использовать общие страницы, если это предусмотрено способом подготовки. Этот вариант и влияние страниц на память разберём в разделе 6.4.

Один подготовленный шрифт используем для множества надписей. Новый экземпляр не нужен для каждой кнопки, имени NPC или строки меню.

Освобождение ресурсов

В обоих показанных вариантах шрифт создан напрямую и принадлежит нашему экрану или системе ресурсов. Когда он окончательно перестаёт использоваться, вызываем:

```
font.dispose();
```

Освобождение генератора и освобождение готового шрифта — разные операции. В примере с generateFont() без общего PixmapPacker шрифт владеет созданными текстурными страницами и освобождает их при dispose().

Если BitmapFont загружен через AssetManager, применяем правила раздела 5.4: вручную его не уничтожаем, а используем unload() или освобождение менеджера. Экран, который только получил общий шрифт, также не должен уничтожать его при закрытии.

Теперь мы умеем получить собственный шрифт двумя способами и вывести русский текст. Следующий шаг — собрать свой FontGenerator, чтобы размеры, оформление и повторное использование шрифтов были организованы в одном месте.

6.2. Свой FontGenerator

В предыдущем разделе мы рассмотрели два способа получить BitmapFont: загрузить готовый растровый шрифт или создать его из TrueType во время работы приложения. Для своего интерфейса я использую первый вариант: заранее подготавливаю несколько размеров в Hiero, а в игре выбираю нужный файл.

Эту работу удобно собрать в небольшом классе FontGenerator. Он загружает шрифт, устанавливает цвет и масштаб, а также выбирает размер по умолчанию. Так настройки и правила именования файлов находятся в одном месте.

Что делает этот класс

Наш FontGenerator работает с готовыми .fnt и PNG-страницами. Это авторская фабрика, которая создаёт настроенный BitmapFont. Её не нужно путать с библиотечным FreeTypeFontGenerator из раздела 6.1: растеризацию .ttf мы уже выполнили при подготовке ресурсов.

Для примера используем Montserrat SemiBold. В каталоге assets/fonts должны находиться экспортированные варианты:

```
Montserrat-SemiBold_14.fnt
Montserrat-SemiBold_18.fnt
Montserrat-SemiBold_20.fnt
Montserrat-SemiBold_24.fnt
Montserrat-SemiBold_26.fnt
Montserrat-SemiBold_28.fnt
Montserrat-SemiBold_32.fnt
Montserrat-SemiBold_36.fnt
```

Рядом размещаем все PNG-страницы, на которые ссылается каждый .fnt. Имя файла содержит размер, поэтому путь можно получить по одному правилу. В коде путь начинается с fonts/, без assets/. Экспорт в Hiero разберём в разделе 6.4.

Почему шрифт нужно подготавливать белым

Одноцветные шрифты, которым будем задавать цвет в коде, всегда подготавливаем с белыми символами на прозрачном фоне. При экспорте в Hiero выбираем белую заливку: тогда один и тот же шрифт можно исполь-

зовать для белых, красных, зелёных и других надписей.

В стандартной отрисовке LibGDX цвет, заданный через font.setColor(), умножается на цвет символов в текстуре. Белый цвет не меняет заданную окраску, поэтому белые символы принимают нужный цвет:

```
font.setColor(Color.GREEN);
font.draw(batch, "Уровень 5", x, y);
```

Цвет, установленный через setColor(), сохраняется для последующих прямых вызовов font.draw(). Перед другой надписью задаём нужный ей цвет.

Если же символы изначально чёрные, обычный setColor() их не перекрасит: нулевые значения RGB при умножении останутся нулевыми. Даже Color.WHITE не сделает такой шрифт белым. Для нашей фабрики сохраняем белую основу в PNG-страницах, а цвет надписей выбираем уже в игре. Произвольные цветные символы тоже нельзя превратить в любой нужный цвет таким умножением.

Реализация FontGenerator

Создадим отдельный класс в модуле core. Сохраним перегрузки для размера, масштаба и цвета. В конструктор передадим minScreenSize — меньшую сторону окна, по которой

наш интерфейс выбирает размер из подготовленного набора.

```java
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.Texture.TextureFilter;
import com.badlogic.gdx.graphics.g2d.BitmapFont;

public final class FontGenerator {
    // Набор для экспорта в Hiero, а не для загрузки .fnt.
    public static final String FONT_CHARS =
        "абвгдеёжзийклмнопрстуфхцчшщъыьэюя"
        + "abcdefghijklmnopqrstuvwxyzçğıiöşüñáéíóú"
        + "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
        + "ABCDEFGHIJKLMNOPQRSTUVWXYZÇĞİÖŞÜ"
        + "0123456789][_!$%#@|\\/?-+=()*&.;:,{}\"´`'<>¡¿"
        + " ÑÁÉÍÓÚ—–…№";

    private final float minScreenSize;

    public FontGenerator(float minScreenSize) {
        if (!(minScreenSize > 0f)) {
            throw new IllegalArgumentException( "Invalid screen size");
        }
        this.minScreenSize = minScreenSize;
    }

    public BitmapFont generateDefaultBitmapFont(
            float scale, Color color) {
        return generateBitmapFont(getDefaultFontSize(), scale, color);
    }

    public BitmapFont generateBitmapFont(int size) {
        return generateBitmapFont(size, 1f, Color.WHITE);
    }

    public BitmapFont generateBitmapFont(int size, float scale) {
        return generateBitmapFont(size, scale, Color.WHITE);
    }

    public BitmapFont generateBitmapFont(int size, Color color) {
        return generateBitmapFont(size, 1f, color);
    }

    public BitmapFont generateBitmapFont(
            int size, float scale, Color color) {
        if (size <= 0 || !(scale > 0f) || color == null) {
            throw new IllegalArgumentException( "Invalid font settings");
        }
```

FontGenerator — продолжение листинга

```
        BitmapFont font = new BitmapFont(
            Gdx.files.internal(getFontFilename(size)), false
        );
        font.setColor(color);
        font.getData().setScale(scale);

        for (int i = 0; i < font.getRegions().size; i++) {
            font.getRegions().get(i).getTexture().setFilter(
                TextureFilter.Linear, TextureFilter.Linear
            );
        }
        return font;
    }

    private String getFontFilename(int size) {
        return "fonts/Montserrat-SemiBold_" + size + ".fnt";
    }

    private int getDefaultFontSize() {
        if (minScreenSize <= 300) return 14;
        if (minScreenSize <= 400) return 18;
        if (minScreenSize <= 720) return 20;
        if (minScreenSize <= 1080) return 24;
        if (minScreenSize <= 1280) return 26;
        if (minScreenSize <= 1600) return 28;
        if (minScreenSize <= 1920) return 32;
        return 36;
    }
}
```

Параметр false в конструкторе BitmapFont отключает переворот шрифта по вертикали. Такой вариант подходит для обычного интерфейса LibGDX с осью Y, направленной вверх.

Для сглаженного текста включена фильтрация Linear у всех страниц. Если используется пиксельный шрифт, выбираем Nearest.

Фабрика составляет имя файла по запрошенному размеру. Поэтому generateBitmapFont(32) загружает вариант _32.fnt, а вызов с размером 30 потребует отдельный файл _30.fnt. Класс не генерирует отсутствующий размер и не выбирает ближайший автоматически.

Константа FONT_CHARS нужна при подготовке файлов в Hiero. Конструктор BitmapFont её не использует: он загружает только символы, уже записанные в .fnt. Если добавить в строку новый знак, нужно повторно экспортировать необходимые размеры шрифта. Исходная гарнитура тоже должна содержать этот знак.

Использование

При подготовке интерфейса после инициализации приложения создаём фабрику и несколько шрифтов. Этот фрагмент использует Gdx, BitmapFont и Color, импортированные выше; полученные шрифты сохраняем в полях владельца:

```
FontGenerator fonts = new FontGenerator(
    Math.min(Gdx.graphics.getWidth(), Gdx.graphics.getHeight())
);

BitmapFont bodyFont = fonts.generateDefaultBitmapFont( 1f, Color.WHITE );
BitmapFont titleFont = fonts.generateBitmapFont(32);
BitmapFont hintFont = fonts.generateBitmapFont(18, Color.GRAY);
```

Если игра уже рассчитывает minScreenSize, передаём эту величину. Пороговые значения — правило конкретного интерфейса, а не универсальная формула для всех устройств. При фиксированном UI-viewport размер дополнительно согласуем с его масштабом; это разберём в разделе 6.3.

Здесь minScreenSize сохраняется при создании фабрики. При смене профиля интерфейса можно создать новую фабрику или явно выбрать другой размер через generateBitmapFont(size). Уже загруженные шрифты от этого автоматически не меняются.

Создаём один раз и используем повторно

Каждый вызов generateBitmapFont() возвращает новый объект и заново загружает его текстурные страницы. Даже два обращения к одному .fnt в этом варианте создадут отдельные текстуры.

Поэтому создаём шрифты при загрузке или подготовке экрана и сохраняем ссылки. Один bodyFont можно использовать для множества надписей. Вызывать фабрику в render() или загружать новый шрифт для каждой кнопки не нужно.

Для Label шрифт передаём через стиль. Label импортируем из com.badlogic.gdx.scenes.scene2d.ui:

```
Label.LabelStyle bodyStyle =
    new Label.LabelStyle(bodyFont, Color.WHITE);

Label healthLabel = new Label("Здоровье: 100", bodyStyle);
```

У Label цвет текста определяется цветом элемента и LabelStyle.fontColor. В частности, цвет Color.GRAY, установленный фабрикой у hintFont, сам по себе не задаёт цвет надписи Label: его указываем в стиле или у элемента.

Для другого цвета можно использовать тот же bodyFont, создав отдельный стиль:

```
Label.LabelStyle hintStyle =
    new Label.LabelStyle(bodyFont, Color.GRAY);

Label hintLabel = new Label("Нажмите для продолжения", hintStyle);
```

Новых текстур ради серой надписи здесь не создаётся. Если задаём цвет и в стиле, и у самого Label, при отрисовке они перемножаются. Белый цвет одного из них сохраняет окраску, заданную другим.

При необходимости стиль можно зарегистрировать в уже созданном Skin игрового интерфейса. В следующем фрагменте он доступен через переменную skin; его время жизни должно охватывать все использующие его экраны:

```
skin.add("game-label", bodyStyle, Label.LabelStyle.class);
Label questLabel = new Label("Найти торговца", skin, "game-label");
```

В такой схеме в skin зарегистрирован только LabelStyle: он ссылается на шрифт, но не освобождает его. Шрифт остаётся у прежнего владельца.

Если зарегистрировать сам BitmapFont отдельным ресурсом skin, skin.dispose() будет освобождать и его. Такой вариант выбираем только при передаче владения: тот же шрифт не должен одновременно освобождаться экраном или AssetManager. Подробно устройство интерфейса рассмотрим в разделе 14.1.

Кто освобождает шрифты

Фабрика не хранит созданные объекты и не управляет их временем жизни. Владельцем становится экран или система ресурсов, сохранившая полученный шрифт.

Если экран сам создал эти три шрифта и больше никто их не использует, при окончательном закрытии интерфейса освобождаем их:

```
bodyFont.dispose();
titleFont.dispose();
hintFont.dispose();
```

Общий шрифт игры освобождает общий владелец после всех использующих его экранов. Если владение передано Skin, второй раз вызывать dispose() вручную нельзя.

Объект шрифта изменяемый. При изменении его общего масштаба нужно пересчитать текст и размещение всех использующих его элементов. Если нужно изменить только одну надпись Scene2D, используем Label.setFontScale(); подбор коэффициента и качество результата разберём в разделе 6.3.

Значительное увеличение готовых глифов не добавляет деталей. Для крупного текста подготавливаем файл большего размера.

В результате у нас есть небольшая фабрика с понятной задачей: выбрать подготовленный ресурс, загрузить его и настроить. Она подходит для настольной, Android- и браузерной версий, потому что во время работы использует обычный BitmapFont без нативного FreeType.

6.3. Размеры и качество текста

Шрифт может быть правильно загружен, содержать кириллицу и при этом плохо выглядеть в игре: буквы слишком мелкие, заголовок размытый, а кнопка не вмещает текст. Здесь важны не только параметры генерации, но и масштаб интерфейса.

Сначала выберем роли текста, фильтрацию и способ размещения. Затем разберём более точный подбор разрешения под viewport и HiDPI. Для этого углубления понадобятся камера и viewport из главы 7; при первом чтении к нему можно вернуться после знакомства с ними.

Размер глифов и размер на экране

Различаем три величины:

• размер растеризации — размер, выбранный при экспорте в Hiero или

при генерации через FreeType;

• размер в координатах интерфейса — сколько места занимает текст в его

макете;

• размер на устройстве — сколько физических пикселей приходится на

этот текст после применения viewport и масштаба.

Например, интерфейс высотой 270 единиц выводится в область высотой 720 физических пикселей. На одну единицу приходится примерно 2.67 пикселя. Шрифт, подготовленный в размере 24 и нарисованный без дополнительного масштаба, заметно увеличится на экране. В физических пикселях надпись станет крупнее, но новых деталей в изображениях букв не появится. В макете она по-прежнему занимает то же место в единицах UI.

Поэтому изменение camera.zoom или setFontScale() не заменяет генерацию глифов нужного разрешения. Масштаб меняет размер готовых изображений.

Начинаем с нескольких ролей текста

Для интерфейса с областью порядка 480 × 270 единиц можно начать с таких ориентиров:

Роль	Примерный размер в единицах UI
Второстепенная подпись	14–16
Основной текст и кнопки	18–24
Заголовок	28–36

Это стартовые значения для проверки макета, а не универсальные размеры для любого телефона. Гарнитуры отличаются высотой букв, шириной и толщиной линий. Прове-

ряем текст на устройстве, в обычном положении и на настоящем фоне игры.

Важное сообщение не должно требовать вглядывания. Если текст не помещается, сначала проверяем ширину элемента, перенос строк и формулировку. Уменьшать весь интерфейс ради одной длинной кнопки обычно неудачное решение.

Фильтрация и масштабирование

Для обычного сглаженного текста используем Linear: при умеренном изменении размера он сглаживает выборку текстуры. Но маленький шрифт, растянутый в четыре раза, всё равно потеряет чёткость. Фильтрация не восстанавливает отсутствующие детали.

Для пиксельного шрифта часто подходит Nearest и целочисленный масштаб. Так рисунок пикселей сохраняется. Если применять дробный масштаб, отдельные линии могут получаться разной толщины.

У BitmapFont есть настройка округления координат:

```
bodyFont.setUseIntegerPositions(false);
```

Если шрифт будет использоваться в Label, задаём эту настройку до создания надписей: у них собственные кэши тек-

ста. Изменение настройки готового шрифта не обновляет уже созданные кэши автоматически.

Она может помочь сглаженному тексту при дробном масштабе, когда округление в единицах UI вызывает неровное размещение. Для пиксельной графики обычно важнее выравнивание по физической пиксельной сетке. Эта настройка не заменяет правильный размер и фильтр.

Измеряем текст, а не угадываем ширину

Одинаковое количество букв не означает одинаковую ширину строки. Для измерения используем GlyphLayout, а в Scene2D обычно доверяем расчёт Label и Table.

Для неизменного заголовка создаём layout один раз. titleFont уже подготовлен по разделу 6.2, а prepareTitle() вызывается после его создания:

```
import com.badlogic.gdx.graphics.g2d.GlyphLayout;

// Поле и метод экрана:
private final GlyphLayout titleLayout = new GlyphLayout();

private void prepareTitle() {
    titleLayout.setText(titleFont, "Настройки");
}
```

В открытом проходе batch можно нарисовать уже рассчитанный текст по центру. Здесь uiViewport обновлён, камера

интерфейса расположена в центре его области, а её zoom равен 1:

```
titleFont.draw(batch, titleLayout,
    (uiViewport.getWorldWidth() - titleLayout.width) / 2f, 220f);
```

Если текст или параметры шрифта изменились, пересчитываем layout. Новый объект при каждой отрисовке создавать не нужно.

Подбор разрешения под viewport и HiDPI

Если готовый текст заметно увеличивается при выводе на экран, можно взять глифы большего разрешения и уменьшить их масштаб в координатах UI. Так размеры макета сохранятся, а изображение букв будет подробнее.

Этот пример продолжает использование фабрики из раздела 6.2. Дополнительно экспортируем размеры 64 и 128: в assets/fonts нужны файлы Montserrat-SemiBold_64.fnt, Montserrat-SemiBold_128.fnt и все их PNG- страницы. В качестве профилей основного текста используем подготовленные размеры 24, 32, 36, 64 и 128.

Ниже uiViewport уже создан и обновлён по размеру окна. Его камера — ортографическая, без поворота, с положительным zoom; размеры окна и viewport также положительны. Расчёт выполняем при подготовке интерфейса, а не каж-

дый кадр. Помимо классов из 6.2, понадобятся импорты:

```
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.glutils.HdpiUtils;
```

В методе подготовки выбираем файл и масштаб:

```
OrthographicCamera uiCamera =
    (OrthographicCamera) uiViewport.getCamera();

float pixelsPerUiUnit =
    HdpiUtils.toBackBufferY(uiViewport.getScreenHeight())
    / (uiViewport.getWorldHeight() * uiCamera.zoom);

float textSizeInUi = 24f;
int desiredRasterSize = Math.max( 1,
    (int) Math.ceil(textSizeInUi * pixelsPerUiUnit)
);

int[] preparedSizes = {24, 32, 36, 64, 128};
int rasterSize = preparedSizes[preparedSizes.length - 1];
for (int preparedSize : preparedSizes) {
    if (preparedSize >= desiredRasterSize) {
        rasterSize = preparedSize;
        break;
    }
}

float fontScale = textSizeInUi / rasterSize;
BitmapFont bodyFont = fonts.generateBitmapFont(
    rasterSize, fontScale, Color.WHITE
);
```

HdpiUtils переводит высоту экранной области viewport в физические пиксели буфера окна — backbuffer. Учитываем высоту именно области вывода интерфейса, а не всего окна: viewport может оставлять полосы по краям.

Выбираем наименьший подготовленный размер, который не меньше расчётного. Если такого нет, используем самый большой из набора. Это позволяет загрузить существующий файл, но при недостаточном разрешении он всё равно будет увеличиваться. Тогда проверяем результат и при необходимости экспортируем ещё один профиль.

Масштаб считаем по **выбранному** размеру: textSizeInUi / rasterSize. Например, при требуемом разрешении 50 выбираем файл 64, а для номинального размера 24 единицы UI используем масштаб 24 / 64 = 0.375. Поэтому выбор более крупного файла не увеличивает надпись в макете. Высота конкретных букв всё равно зависит от метрик гарнитуры.

При высоте UI 270 единиц, zoom = 1 и номинальном размере текста 24 получаем:

Высота области в физических пикселях	Пикселей на единицу UI	Выбранный размер	Масштаб шрифта в UI
270	1	24	1

Высота области в физических пикселях	Пикселей на единицу UI	Выбранный размер	Масштаб шрифта в UI
720	около 2.67	64	0.375
1440	около 5.33	128	0.1875

Полученный шрифт сохраняем и повторно используем в этой системе UI. Для другой камеры или независимого масштаба выбираем отдельный вариант.

Если используем Label.setFontScale(), учитываем, что он задаёт абсолютный масштаб глифов для надписи, а не множитель текущего масштаба BitmapFont. Чтобы увеличить одну надпись на 20%, передаём fontScale * 1.2f. Например, при исходном масштабе 0.375 получаем 0.45. Разрешение самих глифов от этого не изменится.

Не генерируем шрифт при каждом движении границы окна

Приведённая формула объясняет подбор разрешения, но не означает, что нужно загружать или генерировать новый шрифт на каждом событии resize(). При перетаскивании окна таких событий может быть много.

На практике удобно подготовить несколько профилей: например, обычный и повышенной чёткости. Выбираем подходящий профиль по масштабу, а небольшую разницу компенсируем отображением. Для готовых .fnt используем тот

же принцип: несколько экспортированных размеров вместо одного огромного шрифта на все случаи.

Если всё же заменяем набор шрифтов, сначала подготавливаем новый, затем обновляем стили всех использующих его элементов и только после этого освобождаем старый. Наша фабрика не хранит кэш: загруженные варианты держит владелец интерфейса или система ресурсов. Именно он учитывает живые ссылки и решает, когда можно вызвать dispose().

После замены шрифта в Label вызываем setStyle() с нужным стилем, чтобы обновились его кэш и расчёт размеров. Изменение общего масштаба также требует пересчёта использующих его надписей и макета.

Для большого диапазона масштабов существуют distance field fonts: они используют другое представление глифов и специальный шейдер. Это отдельный подход, а не режим обычного BitmapFont, который достаточно включить одним флагом. Для большинства меню нескольких растровых размеров вполне хватает.

В результате выбираем размер по удобству чтения, разрешение — по масштабу вывода, а размещение — по реальным метрикам текста. В следующем разделе разберём браузерную сборку и стоимость текстурных страниц.

6.4. Шрифты в браузере, текстурные страницы и память

Одинаковый интерфейс должен оставаться читаемым и на телефоне, и в настольном приложении, и в браузере. Но способ подготовки шрифтов может различаться: генерация через нативный FreeType подходит не каждой платформе.

Для нашей фабрики из раздела 6.2 основной путь — экспортировать шрифты и загрузить готовые ресурсы. Общий упаковщик, который рассмотрим ниже, — отдельный вариант для проектов с генерацией во время работы приложения.

Что меняется в GWT-сборке

Стандартный gdx-freetype использует нативную библиотеку. Его нельзя просто добавить к HTML-модулю на GWT и ожидать, что браузер загрузит те же нативные файлы, что и настольное приложение.

Это ограничение способа генерации, а не вывода текста. BitmapFont, готовые .fnt и текстурные страницы работают в браузере. Существуют дополнительные реализации генерации для веба, но они требуют отдельной интеграции и проверки совместимости.

Наш FontGenerator уже использует общий вариант: заранее подготовленные шрифты для всех платформ. Тогда ин-

терфейс использует одинаковые ресурсы, а запуск не включает растеризацию TrueType.

Экспортируем шрифт через Hiero

Hiero — инструмент подготовки растровых шрифтов для LibGDX. Ссылку на загрузку и варианты запуска можно найти в документации Hiero[7]. Он работает отдельно от игры: результат экспорта используем как обычный ресурс.

Последовательность работы такая: 1. Открываем исходный .ttf или выбираем нужную гарнитуру.

[7] документации Hiero: https://libgdx.com/wiki/tools/hiero

2. Задаём размер и набор символов из FontGenerator.FONT_CHARS: латиницу, кириллицу, цифры и используемую пунктуацию. 3. Выбираем способ растеризации. У разных режимов отличаются доступные эффекты и результат сглаживания. 4. Для одноцветного шрифта сохраняем белые глифы на прозрачном фоне, как в разделе 6.2. Проверяем это в предпросмотре. 5. Проверяем текст вроде Задание № 3 — найти ещё 10 монет…, включая Ё и ё. 6. Выбираем размер текстурных страниц и проверяем, что глифы не обрезаны. Одна огромная страница для небольшого набора символов может расходовать память впустую. 7. Сохраняем шрифт в формате BMFont: файл .fnt и все созданные PNG- страницы.

Например, сохраняем файлы Montserrat-SemiBold_24.fnt

и Montserrat- SemiBold_32.fnt в assets/fonts, чтобы они соответствовали правилу именования фабрики. Для выбора размера по умолчанию готовим весь набор из раздела 6.2, а для примера повышенной чёткости из 6.3 — также размеры 64 и 128.

Имена соответствующих страниц берём из экспорта. Не переименовываем PNG отдельно от описания: приложение ищет страницы по записям внутри .fnt. Проверяем результат уже в игре на реальной надписи и нужном масштабе.

Для воспроизводимой подготовки сохраняем также настройки Hiero и исходный шрифт в рабочем каталоге проекта. В игру включаем только ресурсы, которые нужны во время выполнения. Если весь текст уже растеризован, исходный .ttf для его отрисовки не требуется.

Если нужно автоматизировать экспорт, в gdx-tools есть BitmapFontWriter: он позволяет сохранить данные, созданные FreeType, в .fnt и PNG. Это инструмент подготовки ресурсов; помещать его в мобильную или браузерную сборку ради обычной отрисовки не нужно.

Загружаем готовые шрифты через AssetManager

Готовые шрифты можно загружать напрямую через нашу фабрику или через общий менеджер. Для второго варианта добавим задания в GameAssets.queue() из раздела 5.2, рядом

с атласом и аудиофайлами. В GameAssets понадобится импорт com.badlogic.gdx.graphics.g2d.BitmapFont:

```
assets.load("fonts/Montserrat-SemiBold_24.fnt", BitmapFont.class);
assets.load("fonts/Montserrat-SemiBold_32.fnt", BitmapFont.class);
```

После завершения загрузки получаем их:

```
BitmapFont bodyFont = assets.get(
    "fonts/Montserrat-SemiBold_24.fnt", BitmapFont.class
);
BitmapFont titleFont = assets.get(
    "fonts/Montserrat-SemiBold_32.fnt", BitmapFont.class
);
```

Текстурные страницы загружаются как зависимости. В очередь добавляем .fnt, а все указанные внутри него изображения должны присутствовать в ресурсах.

Это альтернативный способ загрузки: для управляемых ресурсов получаем шрифты через assets.get(), вместо повторного создания через нашу фабрику. Надписи и стили работают с ними так же, как с напрямую загруженными шрифтами. Владельцем остаётся AssetManager; вручную вызывать bodyFont.dispose() нельзя.

Если способы подготовки отличаются по платформам

Если у настольной и мобильной версий сохраняется генерация, отделяем её от общего интерфейса. В core можно определить небольшой контракт:

```
import com.badlogic.gdx.graphics.g2d.BitmapFont;

public interface FontProvider {
    BitmapFont getBodyFont();
    BitmapFont getTitleFont();
}
```

Экраны получают такой объект и используют готовые шрифты. Реализация с готовыми файлами возвращает ресурсы из AssetManager, а реализация с растеризацией использует FreeTypeFontGenerator из первого раздела. Для нашей фабрики готовых .fnt разделение по платформам не требуется. Классы, использующие нативный FreeType, размещаем в коде, который не входит в компиляцию GWT, например в отдельном модуле для настольной и Android- версий.

Простая проверка платформы через if не заменяет это разделение: общий код всё равно может содержать недоступные GWT классы. Поставщик выбирает способ подготовки,

а правила освобождения остаются у соответствующего владельца ресурсов.

Почему несколько шрифтов могут занимать много памяти

Отдельные экспорты размера 24 и размера 32 обычно содержат разные текстурные страницы. Прямая загрузка одного .fnt дважды через нашу фабрику также создаёт две отдельные копии текстур в памяти. Ещё два варианта с обводкой добавляют новые изображения тех же символов.

Стоимость зависит не только от размера .ttf или PNG на диске. Для несжатой текстуры RGBA8888 грубая оценка такова:

ширина × высота × 4 байта.

Страница 1024 × 1024 занимает около **4 МиБ** текстурных данных. Несколько страниц суммируются. Mipmap-уровни добавляют примерно треть к объёму базового уровня; кроме того, могут сохраняться CPU-копии изображений и данные шрифта.

PNG может весить намного меньше благодаря сжатию, но это не означает, что его содержимое занимает столько же памяти после загрузки на GPU.

Поэтому сначала ограничиваем число размеров, стилей и символов. Общую упаковку вводим тогда, когда она даёт заметную пользу.

Упаковываем два размера в общие страницы

FreeTypeFontParameter.packer позволяет передать один PixmapPacker нескольким операциям генерации. Тогда изображения глифов размещаются на общих страницах. При заполнении страницы упаковщик создаёт следующую

— один packer не гарантирует одну текстуру.

Ниже — вариант подготовки через FreeType для настольного приложения и Android. Это углубление: для обычной загрузки .fnt такой класс не требуется. Зависимости gdx-freetype уже подключены по разделу 6.1, а исходный шрифт находится в assets/fonts/ui.ttf.

Класс создаёт два размера и владеет общими страницами. Набор символов берём из FontGenerator.FONT_CHARS из раздела 6.2. Генерацию выполняем после инициализации приложения в потоке с графическим контекстом. Для браузера эти ресурсы нужно экспортировать заранее: показанный конструктор создаёт шрифты в памяти и сам не пишет .fnt или PNG.

```
import com.badlogic.gdx.files.FileHandle;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.Pixmap;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.Texture.TextureFilter;
```

SharedFonts — продолжение листинга

```
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.PixmapPacker;
import com.badlogic.gdx.graphics.g2d.freetype
    .FreeTypeFontGenerator;
import com.badlogic.gdx.graphics.g2d.freetype
    .FreeTypeFontGenerator.FreeTypeFontParameter;
import com.badlogic.gdx.utils.Array;
import com.badlogic.gdx.utils.Disposable;

public final class SharedFonts implements Disposable {
    private final PixmapPacker packer = new PixmapPacker(
        1024, 1024, Pixmap.Format.RGBA8888, 2, false
    );
    private final BitmapFont[] fonts = new BitmapFont[2];
    private boolean disposed;

    public SharedFonts(FileHandle file) {
        try {
            FreeTypeFontGenerator generator =
                new FreeTypeFontGenerator(file);
            try {
                FreeTypeFontParameter parameter =
                    new FreeTypeFontParameter();
                parameter.characters = FontGenerator.FONT_CHARS;
                parameter.color = Color.WHITE;
                parameter.incremental = false;
                parameter.packer = packer;
                parameter.minFilter = TextureFilter.Linear;
                parameter.magFilter = TextureFilter.Linear;

                parameter.size = 24;
                fonts[0] = generator.generateFont(parameter);
                parameter.size = 32;
                fonts[1] = generator.generateFont(parameter);
            } finally {
                generator.dispose();
            }
        } catch (RuntimeException | Error failure) {
            dispose();
            throw failure;
        }
    }

    public BitmapFont getBodyFont() {
        checkActive();
```

SharedFonts — продолжение листинга

```
        if (disposed) {
            throw new IllegalStateException("Fonts are disposed");
        }
    }

    @Override
    public void dispose() {
        if (disposed) {
            return;
        }
        disposed = true;

        for (int i = 0; i < fonts.length; i++) {
            if (fonts[i] != null) {
                fonts[i].dispose();
            }
        }

        Array<PixmapPacker.Page> pages = packer.getPages();
        for (int i = 0; i < pages.size; i++) {
            Texture texture = pages.get(i).getTexture();
            if (texture != null) {
                texture.dispose();
            }
        }
        packer.dispose();
    }
}
```

При подготовке интерфейса создаём набор и сохраняем его у владельца. Для следующего фрагмента нужен импорт com.badlogic.gdx.Gdx:

```
SharedFonts sharedFonts = new SharedFonts(
    Gdx.files.internal("fonts/ui.ttf")
);
BitmapFont bodyFont = sharedFonts.getBodyFont();
BitmapFont titleFont = sharedFonts.getTitleFont();
```

Это альтернатива созданию этих двух шрифтов через фабрику из 6.2. Вызовы getBodyFont() и getTitleFont() возвращают подготовленные объекты; повторной генерации не происходит. Их можно передать в стили Label, как раньше.

С общим packer созданные шрифты не владеют его текстурами. Поэтому их dispose() не освобождает общие страницы. В SharedFonts.dispose() сначала освобождаем объекты шрифтов, затем каждую текстурную страницу один раз через список упаковщика и сам packer.

Одного packer.dispose() недостаточно для уже созданных текстур страниц. И нельзя заставлять каждый шрифт владеть общей текстурой через setOwnsTexture(true): освобождение одного шрифта тогда лишит изображений остальные.

Все использующие SharedFonts надписи должны перестать рисоваться до его уничтожения. Затем владелец набора вызывает:

```
sharedFonts.dispose();
```

В этом варианте владение общими страницами остаётся у SharedFonts; не регистрируем их одновременно как ресурсы другого владельца, который тоже будет их уничтожать. Проверка disposed защищает от повторной очистки и получения шрифтов после неё.

Как страницы влияют на renderCalls

Для SpriteBatch важен переход между объектами Texture. Смена BitmapFont сама по себе не обязательно вызывает отправку накопленных данных: если его глифы находятся на той же текстуре, отрисовка может продолжить текущую порцию данных.

Но отдельная текстура шрифта между изображениями интерфейса заставляет batch переключиться на неё, а затем вернуться к текстуре UI. У шрифта с несколькими страницами переключения возможны и внутри одной надписи.

Общие страницы помогают, когда нужные глифы действительно используют одну текстуру и остальные условия отрисовки совместимы. Они не гарантируют один renderCall: влияют также вместимость batch, шейдеры, матрицы и порядок элементов. Нельзя менять порядок интерфейса ради статистики, если это нарушает перекрытие объектов.

Основы работы batch разобраны в разделе 4.2. Измеряем результат на настоящем экране через диагностику из раздела 15.1. Смотрим не только количество шрифтов, но и исполь-

зуемые текстуры, объём памяти и переключения batch.

Теперь у нас есть общий способ подготовки шрифтов для всех платформ и отдельный вариант генерации с общими страницами. В обоих случаях сохраняем нужные размеры, повторно используем готовые объекты и определяем владельца текстур. В следующей главе разберём камеру и viewport

— они управляют тем, как координаты мира и интерфейса превращаются в

изображение на экране.

Глава 7. Камера в 2D-игре

План главы

- 7.1. Координаты игрового мира и экрана
- 7.2. Камера и Viewport
- 7.3. Следование за игроком
- 7.4. Зум камеры
- 7.5. Следование со смещением и стили камеры

7.1. Координаты игрового мира и экрана

Когда мы размещаем персонажа на карте, задаём его положение в игровом мире. Когда игрок нажимает на экран, получаем координаты нажатия в окне приложения. Это разные системы координат.

Пока камера неподвижна, а масштаб совпадает с размером окна, разницу легко не заметить. Но после перемещения камеры или изменения зума прямое использование координат мыши даёт неверный результат.

Координаты мира

Положение игрового объекта не зависит от того, где он сейчас виден на экране. Если персонаж стоит в точке (500, 300), перемещение камеры меняет его изображение на экране, но не позицию в мире.

Мировые единицы выбираем под проект: это могут быть метры, размеры тайлов или единицы, соответствующие пикселям исходной графики. Последний вариант тоже не означает, что одна единица всегда занимает один пиксель окна.

В этой главе используем обычную для 2D-игр систему: X растёт вправо, Y — вверх. Координаты храним в float, чтобы движение не ограничивалось целыми значениями.

Координаты нажатия

LibGDX сообщает положение мыши или касания относительно окна приложения:

```
int screenX = Gdx.input.getX();
int screenY = Gdx.input.getY();
```

Начало этих координат находится в левом верхнем углу, а Y растёт вниз. Они описывают место нажатия, но ещё не говорят, в какую точку карты попал игрок.

Для преобразования передаём координаты в том виде, в котором их вернул API ввода. Дополнительно умножать их на коэффициент HiDPI из предыдущей главы не нужно: он использовался для выбора разрешения текстур шрифта, а здесь работаем с координатами окна.

Из экрана в игровой мир: unproject()

Для преобразования используем viewport игровой камеры. В следующих фрагментах предполагается ортографическая 2D-камера. worldViewport уже создан, обновлён под положительный размер окна, а матрица камеры актуальна. Его настройку разберём в разделе 7.2.

Временные векторы создаём один раз в полях экрана:

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.math.Vector2;

private final Vector2 touchWorld = new Vector2();
private final Vector2 screenPosition = new Vector2();
private final Vector2 labelPosition = new Vector2();
```

Для простого примера с одним указателем обработка нажатия выглядит так:

```
if (Gdx.input.justTouched()) {
    int screenX = Gdx.input.getX();
    int screenY = Gdx.input.getY();

    if (isInsideWorldViewport(screenX, screenY)) {
        touchWorld.set(screenX, screenY);
        worldViewport.unproject(touchWorld);
        player.setMoveTarget(touchWorld.x, touchWorld.y);
    }
}
```

worldViewport имеет тип Viewport, а setMoveTarget() — наш метод назначения цели движения. После unproject() вектор содержит координаты мира, в который смотрит камера этого viewport.

Преобразование учитывает положение и зум камеры, а также размер и расположение области просмотра. Дополни-

тельно переворачивать Y перед вызовом не нужно: это уже делает unproject().

Для FitViewport используем именно worldViewport.unproject(). Простой вызов camera.unproject() без границ viewport предполагает область размером со всё окно и не учитывает оставленные полосы.

Метод изменяет переданный вектор. Если игровая система хранит цель как Vector2, она должна скопировать координаты: при следующем нажатии мы переиспользуем touchWorld.

Проверяем попадание в область viewport

Нажатие на чёрную полосу вокруг области FitViewport тоже можно преобразовать, но оно даст точку за пределами видимой области. В примере выше такие нажатия отсекает наш метод isInsideWorldViewport():

```
private boolean isInsideWorldViewport(int screenX, int screenY) {
    int left = worldViewport.getScreenX();
    int width = worldViewport.getScreenWidth();
    int height = worldViewport.getScreenHeight();
    int top = Gdx.graphics.getHeight()
        - worldViewport.getScreenY() - height;

    return screenX >= left && screenX < left + width
        && screenY >= top && screenY < top + height;
}
```

getScreenY() задаёт нижнюю границу viewport в координатах с осью Y вверх. Для сравнения с вводом вычисляем его верхнюю границу top, отсчитываемую от верха окна.

Например, если в окне высотой 800 viewport занимает 720 единиц по высоте и начинается на 40 от нижнего края, его верхняя граница в координатах ввода тоже равна 40. Нажатия с screenY от 40 до 759 находятся внутри, а полосы сверху и снизу игнорируются.

Это проверка прямоугольника вывода, а не допустимости движения. Проходимость выбранной точки и границы самой карты проверяет игровая логика.

Из мира на экран: project()

Обратное преобразование полезно, когда нужно узнать, где объект отображается в окне:

```
screenPosition.set(player.getX(), player.getY());
worldViewport.project(screenPosition);
```

Теперь screenPosition содержит экранную позицию персонажа. Однако результат project() использует начало в левом нижнем углу и направление Y вверх — в отличие от координат ввода.

Поэтому нельзя напрямую сравнивать результат project() с Gdx.input.getY(). Для выбора объекта обычно проще пре-

образовать нажатие в мир через unproject() и проверить попадание в его игровой контур.

Мир и интерфейс

Для карты и интерфейса часто используются разные камеры. Игровая камера следует за персонажем, а камера интерфейса оставляет кнопки и показатели на своих местах.

Если нужно разместить подпись интерфейса рядом с персонажем, преобразуем его позицию сначала в экранные координаты, затем в координаты Stage. Предполагается, что viewport сцены интерфейса также обновлён под размер окна, а её камера актуальна:

```
labelPosition.set(player.getX(), player.getY());
worldViewport.project(labelPosition);

labelPosition.y = Gdx.graphics.getHeight() - labelPosition.y;
stage.screenToStageCoordinates(labelPosition);

label.setPosition(labelPosition.x, labelPosition.y + 12f);
```

Переворот Y нужен потому, что project() возвращает координаты с осью вверх, а screenToStageCoordinates() принимает экранные координаты ввода с осью вниз. В результате получаем положение в системе интерфейса; отступ 12 тоже задан в единицах UI.

Здесь label — заранее созданная надпись, добавлен-

ная непосредственно через stage.addActor(label). Полученная точка задаёт положение её нижнего левого угла. Для подписи над головой выбираем соответствующую точку привязки в мире и при необходимости учитываем ширину надписи.

Для элемента во вложенной группе нужны координаты этой группы. А надписью внутри Table управляет её макет: простое изменение позиции может быть перезаписано таблицей. Такие варианты интерфейса разберём в разделе 14.1.

Положение такой подписи обновляем после изменения позиции объекта или камеры, чтобы она оставалась связанной с персонажем. Если подпись рисуется вместе с объектами игрового мира, преобразование через Stage не требуется: используем мировые координаты.

Главное правило: **перед использованием координат нужно понимать, к какому пространству они относятся**. Для нажатия на карту используем viewport мира, для интерфейса — его собственную систему координат.

В следующем разделе настроим камеру и viewport и разберём, как сохранять правильное отображение игры на разных экранах.

7.2. Камера и Viewport

Камера определяет, на какой участок игрового мира мы смотрим. Viewport задаёт, как этот участок разместить в окне приложения. Вместе они позволяют показывать одну игру на экранах разных размеров.

Ниже — фрагменты одного игрового экрана. В этом варианте он создаёт собственный SpriteBatch, а renderWorld(batch) рисует подготовленные игровые объекты. Настройка камеры и viewport заменяет прежнюю настройку экрана; второй набор этих объектов для того же прохода не нужен.

OrthographicCamera

Для наших 2D-примеров используем OrthographicCamera. Она даёт ортогональную проекцию: объекты не становятся меньше из-за расстояния до камеры.

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.utils.viewport.ExtendViewport;
import com.badlogic.gdx.utils.viewport.FitViewport;
import com.badlogic.gdx.utils.viewport.Viewport;

// Поля игрового экрана:
private OrthographicCamera camera;
private Viewport worldViewport;
private SpriteBatch batch;
```

Положение camera.position соответствует центру видимой области. При перемещении камеры координаты объектов остаются прежними — меняется только наш взгляд на мир.

Размеры camera.viewportWidth и camera.viewportHeight задаются в мировых единицах. При обычной камере без поворота видимая ширина равна viewportWidth * zoom, а высота — viewportHeight * zoom. Пока оставим zoom = 1f.

Когда используется Viewport, он управляет размерами области камеры. Поэтому после его настройки не нужно отдельно подгонять camera.viewportWidth под разрешение окна.

FitViewport: сохраняем заданную область

Начнём с мира, в котором базовая видимая область имеет размер 960 × 540 единиц:

```java
@Override
public void show() {
    if (camera != null) {
        return;
    }

    camera = new OrthographicCamera();
    worldViewport = new FitViewport(960f, 540f, camera);
    batch = new SpriteBatch();

    camera.position.set(480f, 270f, 0f);
}
```

Проверка в начале show() сохраняет уже созданные объекты при возврате на экран. Их окончательное освобождение покажем ниже.

При переключении через Game.setScreen() после show() вызывается resize(), как в разделе 3.3. В нём настроим размеры viewport и пересчитаем матрицы камеры до первой отрисовки.

FitViewport сохраняет пропорции и заданные размеры области мира. Если соотношение сторон окна отличается от 16:9, по краям останутся полосы.

Например, в окне 1280 × 800 область 960 × 540 займёт область окна размером 1280 × 720. Сверху и снизу останется по 40 единиц в координатах окна. Объекты сохранят форму, а игрок увидит тот же участок мира.

960 × 540 — размер базового обзора при zoom = 1, а не

всей карты. Карта может быть значительно больше.

Размеры окна берём из API LibGDX и resize(). На HiDPI-экране они могут отличаться от физических размеров буфера окна; вручную подменять их размерами backbuffer здесь не нужно. Подбор разрешения шрифта под физические пиксели разобран в разделе 6.3.

Обновление при изменении окна

Передаём viewport актуальные размеры из resize():

```
@Override
public void resize(int width, int height) {
    if (width <= 0 || height <= 0) {
        return;
    }
```

Код к разделу 7.2 — продолжение листинга

```
    worldViewport.update(width, height, false);
}
```

update() рассчитывает экранные границы viewport, задаёт размеры области камеры и применяет результат. Здесь false — значение параметра centerCamera, которое сохраняет по-

ложение камеры. Это удобно для игрового мира: изменение окна не должно возвращать обзор к началу карты.

Значение true перемещает камеру в центр области мира viewport. Для неподвижного интерфейса это часто удобно, а при следовании за персонажем может сбить позицию камеры.

Отрисовка через камеру

Перед рисованием мира применяем его viewport и передаём матрицу камеры в SpriteBatch:

```
@Override
public void render(float delta) {
    if (Gdx.graphics.getWidth() <= 0
        || Gdx.graphics.getHeight() <= 0) {
        return;
    }

    Gdx.gl.glClearColor(0f, 0f, 0f, 1f);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    worldViewport.apply();
    batch.setProjectionMatrix(camera.combined);

    batch.begin();
    renderWorld(batch);
    batch.end();
}
```

apply() устанавливает область вывода OpenGL и обновляет камеру. Поэтому сразу после него матрица camera.combined готова для batch, а отдельный camera.update() в этом фрагменте не требуется.

Если меняем положение или зум камеры и используем её матрицы до следующего apply(), вызываем camera.update() после изменения. Например, это нужно перед преобразованием координат через unproject() из раздела 7.1.

Если на экране есть несколько viewport, перед каждой группой отрисовки применяем нужный. Например, после игрового мира вызываем stage.getViewport().apply() и только затем рисуем stage.draw(). Сам вызов Stage.draw() не заменяет применение области вывода.

Для преобразования ввода используем актуальные матрицы камеры. Если камера переместилась после обработки нажатия, преобразование должно соответствовать тому кадру и состоянию обзора, по которому игрок выбирал объект.

ExtendViewport: показываем больше мира

Если полосы не подходят, в show() вместо строки с FitViewport используем:

```
worldViewport = new ExtendViewport(960f, 540f, camera);
```

В таком варианте заданные размеры становятся минимальными. Viewport сохраняет пропорции объектов и расширяет видимую область по одной оси, заполняя окно.

В том же окне 1280 × 800 игрок увидит область 960 × 600 мировых единиц. Полос не будет, а видимая область станет выше. Здесь zoom = 1, максимальные размеры расширения не заданы. Если ограничить расширение максимальными размерами, при выходе за поддерживаемые пропорции полосы могут появиться снова.

Основные варианты можно сравнить так:

Viewport	Область мира	Пропорции объектов	Полосы при разных пропорциях окна
`FitViewport`	Заданный обзор сохраняется	Сохраняются	Возможны
`ExtendViewport` без максимальных размеров	Обзор расширяется по одной оси	Сохраняются	Нет
`StretchViewport`	Заданный обзор растягивается на окно	Могут искажаться	Нет

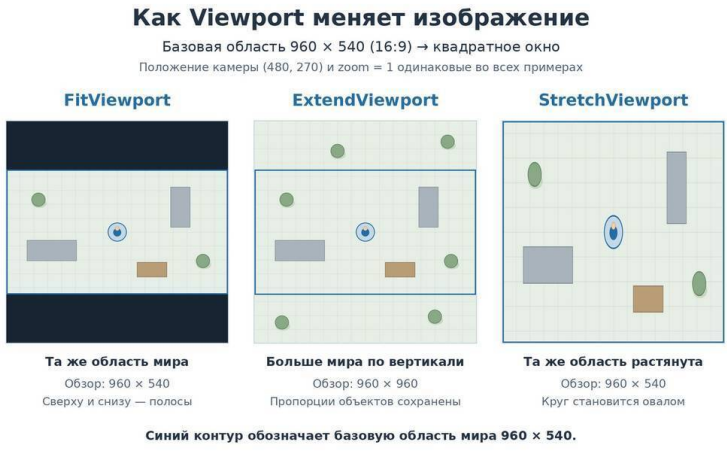


Рисунок 7.1. Одна сцена в квадратном окне при FitViewport, ExtendViewport и

StretchViewport. Положение камеры и zoom одинаковы. Прямоугольный контур отмечает базовую область 960 × 540.

Если важен одинаковый обзор для всех игроков, удобен FitViewport. Если дополнительное пространство по краям допустимо, подходит ExtendViewport. В соревновательной игре расширение обзора может влиять на условия игры — это учитываем в дизайне.

StretchViewport бывает удобен, когда растяжение допустимо, но для мира с персонажами и круглыми объектами обычно нежелательно: круг может стать овалом.

Проверяем настройку

При правильной настройке круг остаётся кругом, нажатие после unproject() попадает в нужный объект, а изменение окна даёт выбранное поведение: полосы или расширение обзора.

В этом примере batch принадлежит экрану. При окончательном удалении экрана освобождаем его:

```
@Override
public void dispose() {
    if (batch != null) {
        batch.dispose();
    }
}
```

После этого экземпляр экрана больше не открываем. Если вместо собственного batch используем общий game.batch из раздела 5.3, его освобождает приложение по правилам 5.4. Камере и viewport вызов dispose() не требуется.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.