ФИЛИПП РАЙДАУТ(PHILIP RIDEOUT)

Начало работы с Filament, рендером на базе физики, на ОС Android

УЧЕБНОЕ РУКОВОДСТВО

Перевод: С. Кузнецов, 2022 г.

Getting Started with Filament on Android

Philip Rideout

Apr 3, 2020 · 9 min

https://medium.com/@philiprideout/getting-started-with-filament-on-android-d10b16f0ec67

Начало работы с Filament, рендером на базе физики, на ОС Android

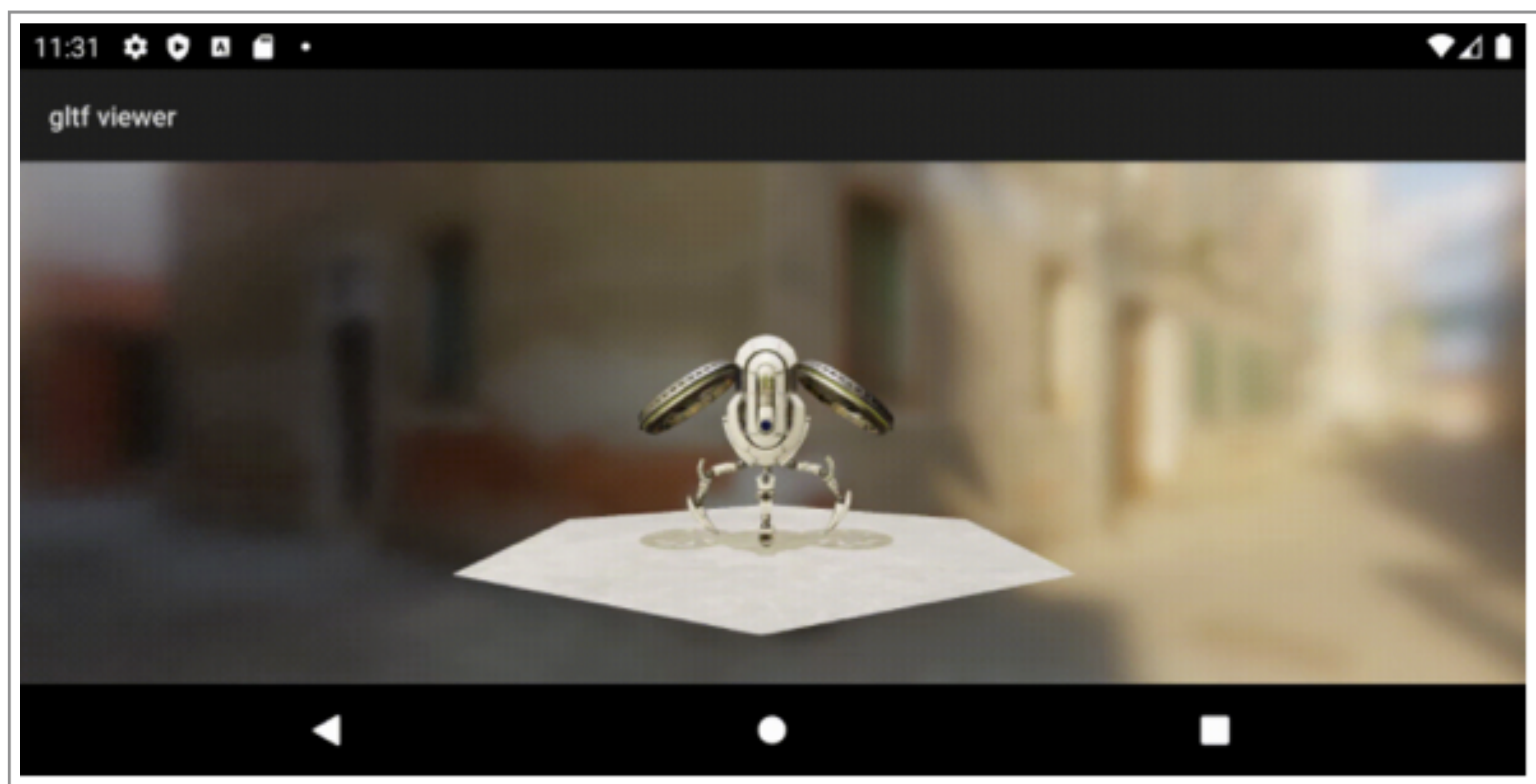
Филипп Райдаут

Апр 3, 2020 · 9 мин

https://medium.com/@philiprideout/getting-started-with-filament-on-android-d10b16f0ec67

Перевод: С. Кузнецов, 08.09.2022

Начало работы с Filament, рендером на базе физики, на ОС Android



Продукт `Filament` является `Google-рендером на базе физики, с открытым исходным кодом(Google's open source physically-based renderer)`. Он замечателен в случае, когда вы должны своему приложению добавить `3D`-возможности, без издержек от целого игрового движка.

Продукт `Filament` может использоваться на множестве платформ (включая ОС `iOS` и `web`), но он особенно подходит к ОС `Android`. В нем есть довольно небольшая центральная библиотека, которая может быть загружена быстро, что очень важно для создания гладкого впечатления в мобильных устройствах.

В этой статье мы рассмотрим процесс создания простого `Android`-приложения, отображающего `glTF 2.0`-модель(подобно модели, показанной

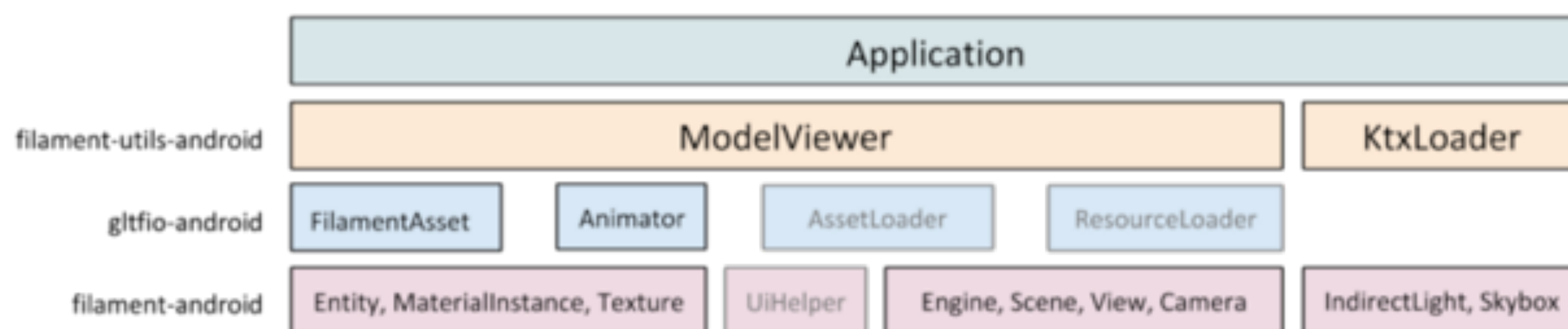
в начале статьи) и позволяющего пользователям панорамировать и масштабировать с помощью знакомых жестов. Результат будет подобен демо-программе вьюера(просмотрщика) **sample-gltf-viewer**, который можете найти в депозитарии продукта **Filament** на сервисе **GitHub** по ссылке [Filament repo on GitHub](#).

1.1 Слои утилит

Далее приведены **слои утилит(utility layers)** продукта **Filament**, доступные с помощью продукта описания проектов **Maven**. В этом учебном руководстве мы будем использовать все эти слои кроме слоя **filamat**.

1. Слой **filament-utils** - высокоуровневые **Android**-утилиты, включая помощников для языка **Kotlin**.
2. Слой **gltfio** - модуль ввода/вывода: материалы и импортеры файлов **glTF 2.0**-модели.
3. Слой **filament** - ядро рендера.
4. Слой **filamat** включает режим генерации материалов во время выполнения.

Следующая схема изображает **Maven**-пакеты, которые мы будем использовать и некоторые классы, предоставляемые ими. Приложения могут использовать, какой-либо уровень абстракции, в котором они нуждаются.



В целом служебные слои не предотвращают взаимодействие с базовыми объектами продукта **Filament**. Например, **FilamentAsset** является просто

неструктурированным контейнером объектов, материалов и текстур. Он самостоятельно предоставляет очень мало функциональности. Точно так же у модуля **`ModelViewer(Просмотрщик модели)`** есть свойства, представляющие актив, движок, сцену, вид и камеру.

Некоторые исключения в этом правиле, показаны серыми контурами на вышеупомянутой схеме, которые скрыты для клиентов модуля **`ModelViewer(Просмотрщик модели)`**.

1.2 Скелет приложения

Для старта, откройте **`IDE`**-среду разработки **`Android Studio`** и создайте новый проект, используя шаблон **`Empty Activity(Пустая активность)`**. В диалоговом окне конфигурации выберите язык программирования **`Kotlin`**. Я надеюсь, что вам нравится язык **`Kotlin`** так же, что я буду использовать, но продукт **`Filament`** поддерживает также языки программирования **`Java`** и **`C++`**. Вы можете выбрать любую минимальную версию набора разработчика **`SDK`** от **`API 19`** или выше.

В корневом уровне откройте файл построения **`build.gradle`** и вверху блока репозитариев добавьте **`mavenCentral()`**. Затем, к своему **`gradle`**-файлу уровня приложения добавьте следующие зависимости.

```
dependencies {
    implementation 'com.google.android.filament:filament-android:1.6.0'
    implementation 'com.google.android.filament:filament-utils-android:
1.6.0'
    implementation 'com.google.android.filament:gltfio-android:1.6.0'
    // ...
}
```

исходник build.gradle находится на сервисе GitHub

В следующий раз, когда **`IDE`**-среда разработки **`Android Studio`** синхронизирует с утилитой **`Gradle`**, она загрузит необходимые архивы с центрального репозитария продукта описания проектов **`Maven`**. Это может

быть вручную инициировано при выборе пункта меню `Sync Project With Gradle Files(Синхронизация проекта с файлами утилиты Gradle)` в меню `File(Файл)` или на панели инструментов.

Затем откройте `MainActivity.kt` и добавьте следующий код.

```
import com.google.android.filament.utils.*

class MainActivity : Activity() {

    companion object {
        init { Utils.init() }
    }

    // ...
```

исходник MainActivity.kt находится на сервисе GitHub

Этот код загружается в виде **«низкоуровневого родного» кода(native code)** для слоя утилит **filament-utils**, который в свою очередь загружает **«низкоуровневый родной» код(native code)** для других двух слоев. Если бы вы использовали только базовые **Filament**-классы, то вы вместо этого сказали бы **Filament.init()**.

В этой точке вы должны иметь возможность выполнить приложение без ошибок.

Далее, добавьте следующие поля к своему классу активности, после сопутствующего объекта, и замените предварительно предоставленный метод создания **onCreate** этим кодом.

```
private lateinit var surfaceView: SurfaceView
private lateinit var choreographer: Choreographer
private lateinit var modelViewer: ModelViewer

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    surfaceView = SurfaceView(this).apply { setContentView(this) }
    choreographer = Choreographer.getInstance()
    modelViewer = ModelViewer(surfaceView)
    surfaceView.setOnTouchListener(modelViewer)
}
```

исходник MainActivity.kt находится на сервисе GitHub

Вы должны будете также добавить некоторые строки импорта **import**, но **IDE**-среда разработки **Android Studio** может сгенерировать их для вас. Затем, давайте позволим продукту **Filament** рендерить во время **обратного вызова кадра(frame callback)**. Добавьте следующие строки к своему **классу активности(действия; activity class)**.

```
private val frameCallback = object : Choreographer.FrameCallback {
    override fun doFrame(currentTime: Long) {
        choreographer.postFrameCallback(this)
        modelViewer.render(currentTime)
    }
}


override fun onResume() {
    super.onResume()
    choreographer.postFrameCallback(frameCallback)
}


override fun onPause() {
    super.onPause()
    choreographer.removeFrameCallback(frameCallback)
}


override fun onDestroy() {
    super.onDestroy()
    choreographer.removeFrameCallback(frameCallback)
}
```

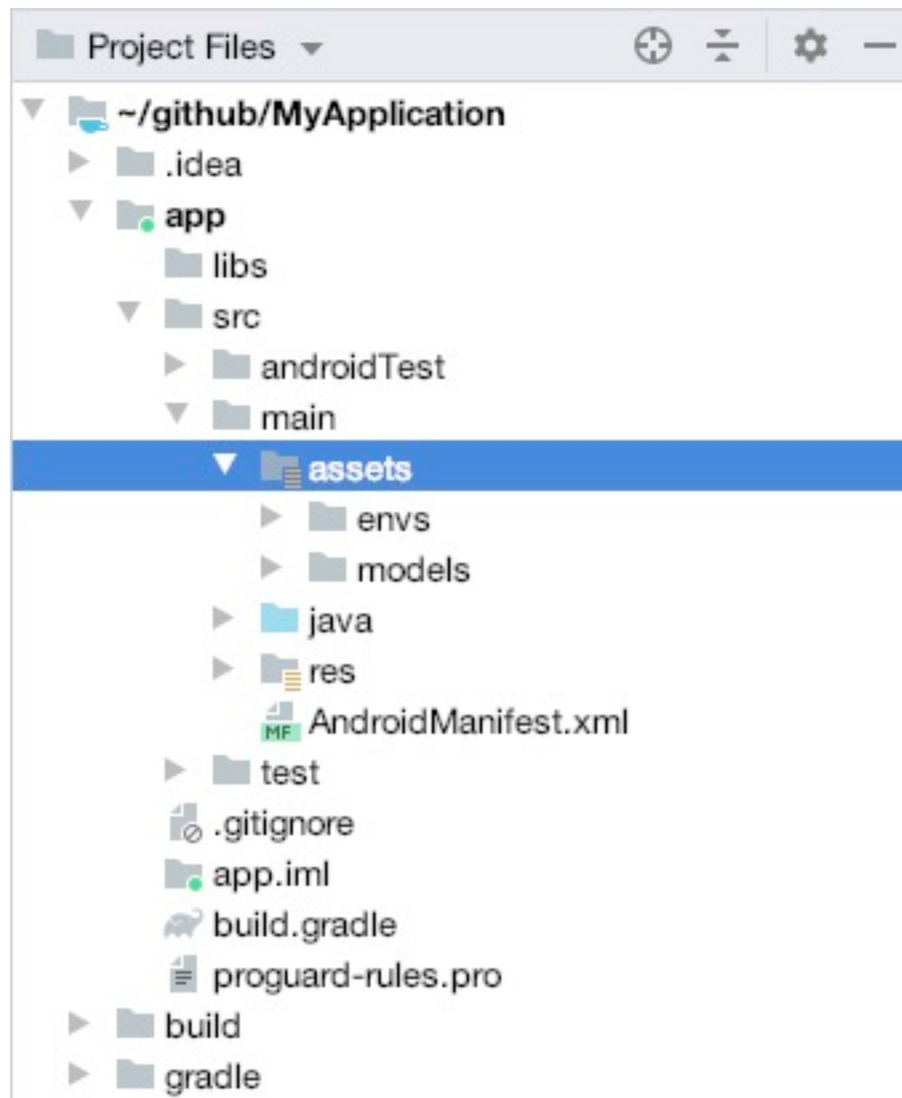
исходник MainActivity.kt находится на сервисе GitHub

В этой точке, если вы выполните приложение, то будет создана инстанция **Filament**-движка и создан **OpenGL**-контекст. В качестве доказательства вы должны в **Android**-журнале видеть что-то вроде этого:

```
I/Filament: FEngine (64 bits) created at 0x79099f2840(I/Filament: FEngine (64 бита) создан в 0x79099f2840)
```

1.3 Добавление активов

Давайте в проект добавим пару `glTF`-моделей, а также некоторые `KTX`-файлы для среды окружения(подробнее о них расскажем позже). Для удобства имеется zip-файл активов и мы использовали его для этого учебного руководства. Вы можете разархивировать их в свой главный каталог(папку) `main` и поэтому итоговая файловая структура похожа на эту нижеуказанную структуру.



Затем сделайте следующие дополнения к главной активности(действию) `MainActivity`. Она использует менеджер активов `AssetManager Android`, чтобы считать содержание `glb`-файла в байтовый буфер `ByteBuffer` и передать его вьюеру(просмотрщику) моделей `ModelViewer`.

```
override fun onCreate(savedInstanceState: Bundle?) {
   // ...
   loadGlb("DamagedHelmet")
   modelViewer.scene.skybox =
Skybox.Builder().build(modelViewer.engine)
}


private fun loadGlb(name: String) {
   val buffer = readAsset("models/${name}.glb")
   modelViewer.loadModelGlb(buffer)
   modelViewer.transformToUnitCube()
}


private fun readAsset(assetName: String): ByteBuffer {
   val input = assets.open(assetName)
   val bytes = ByteArray(input.available())
   input.read(bytes)
   return ByteBuffer.wrap(bytes)
}
```

исходник MainActivity.kt находится на сервисе GitHub

Заметьте вызов функции преобразования в единичный куб `transformToUnitCube()`. Она говорит вьюеру моделей о необходимости преобразовать корневой узел сцены и поэтому он вписывается в куб размером `1 x 1 x 1`, центрируемый в начале координат.

Также добавим поддержку `альбомной ориентации(landscape orientation)`. Откройте файл манифеста `AndroidManifest.xml` и к тегу активности(действию) `<activity>` добавьте следующие два атрибута.

```
android:screenOrientation=
    "fullSensor"
android:configChanges=
    "orientation|screenSize|screenLayout|keyboardHidden
```

Атрибут ориентации экрана **screenOrientation** позволяет **видовому порту(области просмотра; viewport)** поворачиваться во все четыре ориентации, в то время как атрибут изменения конфигурации **configChanges** предусматривает, что активность(действие) должно быть изменено в размерах, а не пересоздано.

В этой точке вы должны безошибочно выполнить **Filament**-приложение и видеть **3D**-модель. **Filament**-приложение должно также отвечать на жесты: **жест падения с одним пальцем(one-finger tumble gesture)**, **панорамирование с двумя пальцами(two-finger pan)** и **изменение масштаба щипком(pinch-to-zoom)**.

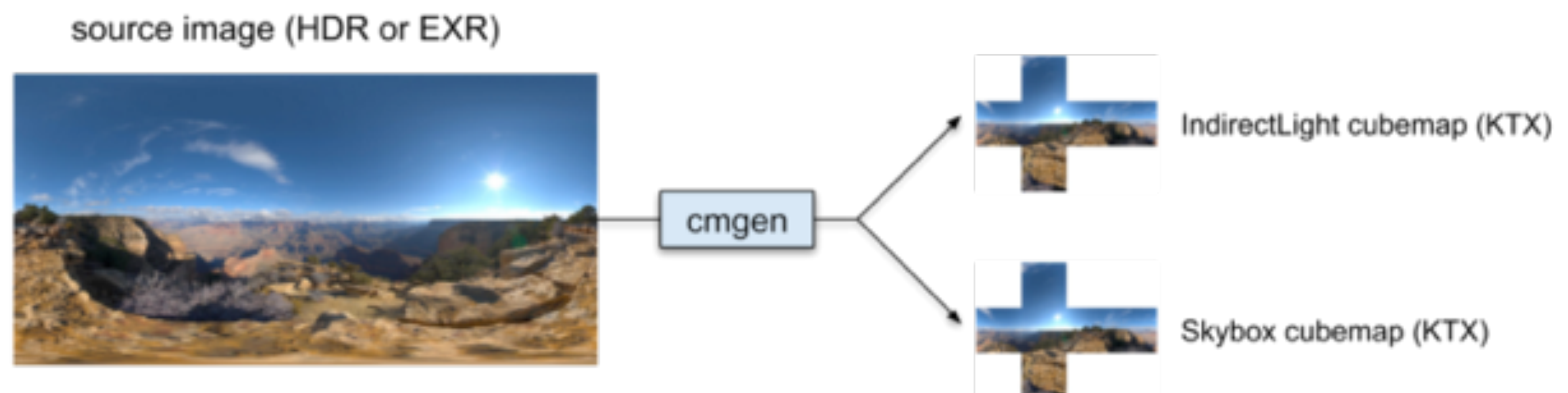
Однако, освещение немного темное. Это вызвано тем, что есть только один источник света, являющийся простым **направленным светом(directional light)**, создаваемым вьюером(просмотрщиком) моделей **ModelViewer**. Чтобы заставить сцену выглядеть намного более приличной, мы должны добавить **источник непрямого света(indirect light source)** и купола с видом неба **skybox**. Это сделаем мы в следующем разделе.

1.4 Свет и небо на базе изображения

Продукт **Filament** поддерживает рендеринг **освещения на базе/основе изображения(image-based lighting)** или **IBL**. Он использует **карту окружающей среды(environment map)** для **аппроксимации освещения по всем направлениям(approximate the lighting all directions)**.

Во время выполнения мы должны создать объект **источник непрямого света(indirect light source)** с именем **IndirectLight**, путем загрузки **KTX**-файла, содержащего набор изображений с плавающей точкой. Вместе эти изображения включают все **уровни множественного отображения(mipmap levels)** и **лицевые стороны карты куба(cubemap faces)**, составляющие окружающую среду. В некотором смысле это **не видимые изображения(not visible images)**; они с большей с точностью в них, поскольку эти данные могут использоваться для **аппроксимации непрямого освещения в сцене(approximate the indirect lighting in the scene)**.

С другой стороны, объект купола с видом неба `Skybox` может быть загружен из `KTX`-файла, который действительно содержит видимые изображения. Продукт `Filament` предоставляет оффлайновый инструмент генерации карт из изображений, с именем `cmgen`, который может использовать `равноугольное изображение(equirectangular image)` и сгенерировать эти два файла одним махом, как изображено ниже.



Пакет актива, который мы уже добавили к проекту, содержит необходимые `KTX`-файлы, поэтому давайте продолжим и добавим некоторый код загрузки их в сцену.(Перейдите к приложению в пункте `1.11` в нижней части статьи, чтобы видеть использование инструмента генерации карт из изображений, с именем `cmgen` для генерации ваших собственных `карт куба(cubemaps)`.)

```
override fun onCreate(savedInstanceState: Bundle?) {
   // ...
   loadGlb("DamagedHelmet")
   loadEnvironment("venetian_crossroads_2k")
}


private fun loadEnvironment(ibl: String) {
 // Создайте источник непрямого света и добавьте его к сцене.
 // Create the indirect light source and add it to the scene.
   var buffer = readAsset("envs/$ibl/${ibl}_ibl.ktx")
   KtxLoader.createIndirectLight(modelViewer.engine,
buffer).apply {
       intensity = 50_000f
       modelViewer.scene.indirectLight = this
   }
 // Создайте купол неба и добавьте его к сцене.
 // Create the sky box and add it to the scene.
   buffer = readAsset("envs/$ibl/${ibl}_skybox.ktx")
   KtxLoader.createSkybox(modelViewer.engine, buffer).apply {
       modelViewer.scene.skybox = this
   }
}
```

исходник MainActivity.kt находится на сервисе GitHub

Теперь, если вы выполняете свое приложение, сцена должна быть намного более захватывающей.

1.5 glTF на базе формата JSON

Загруженная нами модель поврежденного шлема `DamagedHelmet` является бинарным графическим `.glb`-файлом и поэтому все его ресурсы текстуры и буферы вершин встроены в единственный файл. Чтобы загрузить файл с расширением `.gltf`, мы должны будем передать данные в обратном вызове,

которые говорят вьюеру модели ModelViewer о том, как загрузить внешний ресурс из места, по адресу в URI-строке. Попытайтесь внести следующие изменения. Этот код загрузит модель дрона объездчика лошадей BusterDrone(как видно наверху сообщения) вместе с его внешними ресурсами.

```
override fun onCreate(savedInstanceState: Bundle?) {
   // ...

   //loadGlb("DamagedHelmet")
   loadGltf("BusterDrone")
   loadEnvironment("venetian_crossroads_2k")
}


private fun loadGltf(name: String) {
   val buffer = readAsset("models/${name}.gltf")
   modelViewer.loadModelGltf(buffer) { uri ->
readAsset("models/$uri") }
   modelViewer.transformToUnitCube()
}
```

исходник MainActivity.kt находится на сервисе GitHub

Примечание: Модель дрона объездчика лошадей BusterDrone была создана LaVADraGoN и получена из Sketchfab. Она лицензируется под лицензией вида Creative Commons(CC NC).

При попытке выполнить приложение на медленном устройстве или эмуляторе, вы заметите, что отдельные меши постепенно всплывают без предупреждения, в то время как приложение полностью интерактивное. Это происходит из-за асинхронного программного API-интерфейса в загрузчике ресурсов ResourceLoader, позволяющего выполнять декодирование текстур в фоновом режиме.

1.6 Наложение анимации

Библиотека gltfio, ввода/вывода графических gltf-файлов, включает объект аниматора Animator, к которому можно получить доступ через модуль активов FilamentAsset. Чтобы видеть это в действии, попытайтесь заменить свой обратный вызов frameCallback на следующий код.

```
private val frameCallback = object : Choreographer.FrameCallback {
   private val startTime = System.nanoTime()
   override fun doFrame(currentTime: Long) {
       val seconds = (currentTime - startTime).toDouble() /
1_000_000_000
       choreographer.postFrameCallback(this)
       modelViewer.animator?.apply {
           if (animationCount > 0) {
               applyAnimation(0, seconds.toFloat())
           }
           updateBoneMatrices()
       }
       modelViewer.render(currentTime)
   }
}
```

исходник MainActivity.kt находится на сервисе GitHub

Ключевая часть в вышеупомянутом отрывке кода является вызовом функции наложения анимации applyAnimation. Она берет два параметра: индекс(index) в списке определений видов анимаций у модели, и прошедшее время(elapsed time) для той определенной анимации. В glTF-графике, анимации обычно представляют действия. Например, анимация 0 могла бы быть циклом обхода, в то время как анимация 1 является выполняемым циклом.

Начиная с этой модели, используется морфинг(изменение формы; morphing) вместо скиннинга(skinning), поэтому строго не требуется вызов функции обновления матриц костей updateBoneMatrices(). Мы включили его, чтобы показать наиболее успешную практику. В glTF-графике скиннинг(skinning) является ортогональным к анимации(вы можете иметь одно без другого), что является причиной того, почему это не сделано автоматически.

В дополнение к функции наложения анимации applyAnimation и функции обновления матриц костей updateBoneMatrices(), интерфейс аниматора Animator предлагает некоторые простые запросы:

1. int getAnimationCount()
2. float getAnimationDuration(int index)
3. String getAnimationName(int index)

1.7 Погружение глубже

В этой точке, используя только ~100 строк кода, мы создали довольно сложное Android-приложение, которое может рендерить(формировать), анимировать и кувыркать 3D-сцену. Однако, продукт Filament является намного большим, чем просто вьюер(средство просмотра) glTF-файлов, поэтому давайте погрузимся немного глубже и поиграем с частью его базового программного API-интерфейса для разработки.

Отдельные объекты, включаемые в Filament-сцену, являются частью системы типа объект-компонент(entity-component system; ECS). Это позволяет средству рендеринга эффективно делать обход объектов(traverse) сцены способом, ориентированным на данные(data-oriented way) и позволяет делать композицию поведений и атрибутов(composition of behaviors and attributes) без громоздкой иерархии классов.

Продукт Filament не предоставляет тип "узел"("node"), как это имеется в классическом графе сцены, вместо этого продукт Filament предоставляет поддающиеся преобразованию компоненты(transformable components), которые могут быть составлены в дерево(composed into a tree). Таким образом, для каждого узла в glTF-иерархии, gltfio-загрузчик создает объект(entity), и к

каждому из этих объектов он добавляет `поддающийся преобразованию компонент(transformable component)`. Кроме того, если у какого-либо `glTF`-узла есть ассоциированный мешь, то загрузчик добавляет `поддающийся рендерингу компонент(renderable component)` к этому соответствующему объекту.

1.8 Установка преобразования

Чтобы проиллюстрировать использование `системы типа объект-компонент(entity-component system; ECS)`, давайте изменим приложение, чтобы заставить дрон постоянно крутиться вокруг `Z`-оси. Для достижения этого эффекта, мы должны будем захватить `преобразование корневого объекта(transform of the root entity)`.

Примечание: `Корень(root)` - единственный `поддающийся преобразованию объект(transformable entity)`, который не соответствует определенному узлу в `glTF`-файле. Он создается загрузчиком, чтобы позволить преобразование целого актива.

Добавьте следующий отрывок к внутренней части вашей функции `doFrame`.

```
// Сбросьте корневое преобразование, затем поверните его вокруг Z-оси.
// Reset the root transform, then rotate it around the Z axis.
modelViewer.asset?.apply {
    modelViewer.transformToUnitCube()
    val rootTransform = this.root.getTransform()
    val degrees = 20f * seconds.toFloat()
    val zAxis = Float3(0f, 0f, 1f)
    this.root.setTransform(rootTransform * rotation(zAxis, degrees))
}
```

исходник MainActivity.kt находится на сервисе GitHub

Woops, это не будет построено; корневой объект не выставляет(на использование) методы getTransform и setTransform! Поэтому объекты в продукте Filament являются просто целыми числами. Помните, что в системе ECS, объекты не являются объектами со строгим контролем типов(strongly-typed objects). Мы должны из корня извлечь поддающийся преобразованию компонент и использования его вместо этого. К своему классу добавьте следующие два вспомогательных метода. Они используют функции Kotlin-расширения, чтобы позволить использовать более естественный синтаксис, чем предоставленный системой ECS низкоуровневый синтаксис.

```
private fun Int.getTransform(): Mat4 {
    val tm = modelViewer.engine.transformManager
    return Mat4.of(*tm.getTransform(tm.getInstance(this),
null))
}

private fun Int.setTransform(mat: Mat4) {
    val tm = modelViewer.engine.transformManager
    tm.setTransform(tm.getInstance(this), mat.toFloatArray())
}
```

исходник MainActivity.kt находится на сервисе GitHub

Теперь приложение может быть построено и выполнено, и дрон должен крутиться вокруг z-оси.

1.9 Имена, материалы и видимость

В качестве другого примера использования системы типа объект-компонент(entity-component system; ECS), давайте попытаемся скрыть диск пола и отключить эмиссионные задние фары сзади дрона. В конце метода onCreate добавьте следующий код.

```
val asset = modelViewer.asset!!
val rm = modelViewer.engine.renderableManager
for (entity in asset.entities) {
   val renderable = rm.getInstance(entity)
   if (renderable == 0) {
       continue
   }
   if (asset.getName(entity) == "Scheibe_Boden_0") {
       rm.setLayerMask(renderable, 0xff, 0x00)
   }
   val material = rm.getMaterialInstanceAt(renderable, 0)
   material.setParameter("emissiveFactor", 0f, 0f, 0f)
}
```

исходник MainActivity.kt находится на сервисе GitHub

У некоторых `объектов(entities)` в активе нет `поддающегося рендерингу компонента(renderable component)` и поэтому наверху цикла мы проверяем на нуль. Чтобы скрыть диск пола, мы проверяем каждое имя объекта на совпадение со строкой известного имени(это имя художник дал этому определенному объекту; прим.перевод.: `Scheibe_Boden_0`), и в случае совпадения имени вызываем метод установки маски слоя `setLayerMask`, чтобы скрыть его из вида.

`Маска слоя(layer mask)` на `поддающемся рендерингу компоненте(renderable component)` работает в тандеме с `маской видимости(visibility mask)`, установленной в классе вида `View`. Метод установки маски слоя `setLayerMask` берет две `битовых маски(bitmasks)`: список битов для воздействия и заменяющие значения для тех битов. В этом случае мы хотим скрыть `поддающийся рендерингу компонент(renderable component)` от всего и поэтому мы обнуляем все биты `видимости(visibility)`.

Примечание: `Возможность маскирования слоя(layer masking facility)` в продукте `Filament` предназначается только для простых вариантов использования. Другой способ скрыть объект,

состоит в вызове метода удаления объекта из сцены `scene.removeEntity`, который сработал бы только после того, как прогрессивная загрузка завершена полностью. В этом случае мы используем метод установки маски слоя `setLayerMask`, потому что мы хотим скрыть его вскоре после вызова метода загрузки `Gltf`-модели `loadModelGltf`.

Чтобы отключить эмиссионные красные задние фары, мы вызываем метод установки параметра `setParameter` на первом примитиве каждого меша в активе. Это говорит продукту `Filament` о том, чтобы изменять значение параметра материала(также известного как `униформа шейдера(shader uniform)`), и в этом случае мы `значения трех, красный-зеленый-синий, компонентов цвета(red-green-blue values)`, в эмиссионной тройке цвета, устанавливаем в ноль. Вы можете также использовать метод установки параметра `setParameter`, чтобы поменять местами одну текстуру на другую, путем использования одной из следующих строк параметра.

1. `baseColorMap`
2. `metallicRoughnessMap`
3. `normalMap`
4. `occlusionMap`
5. `emissiveMap`

Чтобы видеть весь список параметров `glTF`-материалов, в исходном дереве продукта `Filament` посмотрите ветвь `ubershader.mat.in`.

1.10 Дополнительные материалы для чтения

Вот некоторые ресурсы для бесстрашных разработчиков, желающих погрузиться глубже в тему:

1. На странице проекта `Filament` на сервисе `GitHub` по ссылке [GitHub project page] есть файл `README`, предоставляющий полный обзор с большим количеством ссылок и изображений.

2. Документ проекта **Filament** о **рендере на базе физики(Physically-Based Renderer; PBR)** по ссылке PBR document предоставляет всестороннее объяснение того, как работает **затенение на базе физики(physically-based shading)**.

3. Чтобы создать ваши собственные материалы, прочтите раздел **«Создание пользовательских материалов»(«Creating a custom materials»)** в статье **Бена Доэрти(Ben Doherty)** на сервисе **Medium** по ссылке Ben Doherty's medium article , а также читайте официальный документ **Materials** проекта **Filament** по ссылке Materials document.

1. 11 Приложение: Создание KTX-файлов

В учебном руководстве мы предоставили готовый файл **освещения на базе/основе изображения(image-based lighting)** или **IBL**. Чтобы сделать ваш собственный файл, вы можете получить инструмент **cmgen**(оффлайновый инструмент генерации карт из изображений, с именем **cmgen**, который может использовать **равноугольное изображение(equirectangular image)** и сгенерировать эти два файла одним махом), загружая двоичный пакет для вашей платформы хоста от страницы выпусков **Filament** на **GitHub**, а именно releases page. Обязательно выберите версию, которая соответствует пакет **Maven**, используемому вашим приложением. Двоичный пакет также содержит другие инструменты, такие как **matc**, наш **компилятор материалов(material compiler)**.

Чтобы генерировать и купол неба **Skybox** и освещение **IBL**, вызовите инструмент **cmgen** с использованием командной строки, как в этом примере:

```
cmgen                    \
    --deploy ./myOutDir \
    --format=ktx         \
    --size=256           \
    --extract-blur=0.1  \
    mySrcEnv.hdr
```

Опция размытости извлечения `extract-blur` говорит инструменту `cmgen` о том, что делать купол неба `skybox` в дополнение к освещению `IBL`. Чтобы видеть полный список опций, попробуйте вызвать инструмент с опцией `cmgen -h`.