ИНФОРМАТИКА

Вестник Сыктывкарского университета.
Серия 1: Математика. Механика. Информатика.
Выпуск 4 (41). 2021

УДК 004.057.5 **DOI: 10.34130/1992-2752_2021_4_50**

РЕШЕНИЕ ПРОБЛЕМЫ АБСТРАГИРОВАНИЯ ОТ ПЛАТФОРМОЗАВИСИМОГО КОДА ДЛЯ ПРИЛОЖЕНИЙ IOS И ANDROID НА ПРИМЕРЕ ДВИЖКА SADLION ENGINE

А. В. Ермоленко, В. А. Мельников

В работе рассматриваются существующие решения для кроссплатформенной мобильной разработки, сравниваются их особенности, достоинства и недостатка. Описано решение различных проблем, возникающих при разработке собственного кроссплатформенного движка для разработки под iOS и Android. Рассмотрено построение системы отображения визуального интерфейса на экране пользователя с использованием GPU. Описаны архитектурные решения, применяемые для написания высокопроизводительной логики поведения приложения на языке программирования C++. Рассматриваются жизненные циклы приложений для платформ iOS и Android и предлагается способ абстрагирования от нативного жизненного цикла, для обобщения кода приложения на обеих платформах. Описана реализация межязыкового взаимодействия между Java и C++ посредством JNI на платформе Android и Objective-C и C++, приведены архитектурные решения для построения слоя абстракции, скрывающего такие низкоуровневые взаимодействия в ядре движка.

Ключевые слова: кроссплатформенная разработка, C++, Android, iOS.

На сегодняшний день пользователи запускают приложения на множестве различных платформ, например Windows, Linux, OS X, iOS,

Android, приложение также может работать в веб-браузере и на специальных игровых устройствах, таких как Nintendo Switch, Xbox, Play Station. Для максимально возможного охвата рынка приложение должно работать на наибольшем количестве платформ, разработка при этом не должна усложняться наличием двух и более кодовых баз, которые необходимо одновременно развивать и поддерживать. В дальнейшем игровое направление рассматриваться не будет, и акцент будет сделан именно на разработке приложений.

Автор статьи является основателем и руководителем студии разработки мобильных приложений и игр SadLion Studio, основанной в сентябре 2018 г. (ИП Мельников В. А., ранее ООО «СыкГеймЛаб»), и данная статья является обобщением опыта, полученного при разработке как собственных приложений, так и приложений, выполненных на заказ.

Для разработки кроссплатформенных приложений широко используются готовые движки и фреймворки, например Apache Cordova с использованием языка JavaScript [1] или Xamarin C# [2]. Данные технологии позволяют практически полностью абстрагироваться от работы с нативным кодом и сконцентрироваться именно на разработке приложений, хотя Xamarin C# позволяет вести разработку с использованием библиотек, близких к нативным. Также однозначно в данном подходе стоит отметить, что при увеличении объёма кода скорость сборки JavaScript приложения фактически не изменится, так как он является интерпретируемым. C#, в отличие от JavaScript, является компилируемым языком, но код, написанный на нём, компилируется значительно быстрее кода на C++.

Описанный подход будет работать для простых приложений с небольшим количеством экранов, но при увеличении объёма кода приложения и усложнении его функциональности будут возникать проблемы, связанные с производительностью приложения, например низкая частота кадров анимаций. Подобные проблемы возникали при разработке проекта Big Coloring Book[1] (разработан SadLion Studio) при использовании Xamarin Forms. При отображении на экране списков изображений и блока текстовых тегов с возможностью прокрутки частота

[1]Приложение в магазине Google Play.
URL: https://play.google.com/store/apps/details?id=sadlion.games.bigcoloringbook (дата обращения: 01.12.2021).
Приложение в магазине App Store. URL: https://apps.apple.com/us/app/big-coloring-book/id1475630136 (дата обращения: 01.12.2021).

кадров опускалась ниже 30 кадров в секунду. В статье [3] описывался движок SadLion Engine (SLE), написанный полностью на C++ и использующий для разработки логики C++. Данный движок используется для разработки в SadLion Studio.

При разработке кроссплатформенного движка необходимо решить следующие задачи:

- взаимодействие с нативными функциями и поддержка различной основной архитектуры приложений;
- реализация рендеринга интерфейса и написание логики элементов, расположенных на экране.

Интересное решение поставленных задач приведено в фреймворках Flutter [4], Xamarin Forms [2] и React Native [5]. Так, взаимодействие с нативными функциями во Flutter происходит с помощью отправки сообщений и асинхронного вызова процедур, такой способ напоминает удалённый вызов процедур (RPC). Плюсом такого подхода является то, что Flutter обходит необходимость работы с Java Native Interface [6] (JNI, интерфейс для взаимодействия кода на языках, компилируемых в машинный код, с кодом на Java) в Java на платформе Android, с другой стороны, такое взаимодействие будет медленнее, чем прямой вызов функции, поскольку для реализации ввода-вывода потребуются переключения из контекста пользователя в контекст ядра ОС, например, Unix-сокетов, на основе которых можно такой подход реализовать [7; 8]. Xamarin C# для Android использует схему работы с JNI, а для iOS C# компилируется в машинный код архитектуры ARM и представляет собой обычную библиотеку. React Native использует механизм мостов (bridges) для вызова нативных функций, но запрос для вызова представляет собой JSON-объект. Сериализация и десериализация JSON — довольно трудоёмкий процесс по сравнению с остальными описанными методами.

Архитектуры приложений на iOS и Android отличаются довольно сильно, поэтому каждый фреймворк из описанных выше предоставляет собственную архитектуру приложения. Интерес в данном пункте представляет фреймворк Xamarin, при использовании Xamarin.Forms, используется структура приложения, общая для Android и IOS, за исключением нативных особенностей платформ. Также разработчики могут использовать Xamarin Native, который сильнее разделяет код, и основная разработка ведётся в среде, похожей на нативную [2].

Относительно рендеринга интерфейса особенный интерес представляет фреймворк Flutter. Рендеринг в нём выполняется полностью на GPU, и он не использует стандартные виджеты, но предоставляет собственную библиотеку, соответствующую требованиям стандартного дизайна [4]. Остальные фреймворки, используют нативные элементы интерфейса [2; 5].

Для написания логики поведения элементов каждый из фреймворков предоставляет свой язык программирования, Xamarin — C# [2], React Native — JavaScript [5], Flutter — Dart [4]. Dart компилируется в машинный код для обеих платформ, C# использует виртуальную машину MonoVM на Android и компилируется в машинный код на iOS, JavaScript исполняется в виртуальной машине во всех случаях.

Целью работы является демонстрация архитектурных решений и алгоритмических, применимых для разработки собственного кроссплатформенного фреймворка, использующего не виджеты, а компоненты, на основе которых можно строить виджеты для различных дизайнов, иногда отличающихся от классических для операционных систем, например от Material Design[2].

Дополнительно выдвигаются высокие требования к фреймворку в части отладки с использованием среды Android Studio и XCode, которые поддерживают отладчик LLDB, профилировщики памяти, нагрузки на процессор и энергопотребления. Языком для написания логики был выбран C++, поскольку дизайн языка и его библиотек позволяет писать большие и сложные программные продукты без деградации производительности.

1. Кроссплатформенный рендеринг и написание логики элементов, расположенных на экране

Большинство приложений имеет графический интерфейс. В случае нативных приложений используются компоненты интерфейсов, предоставляемые самой платформой. Данные компоненты соответствуют стилевым рекомендациям самих платформ, так, в случае Android используется стиль Material Design [9; 10], а для iOS — Cupertino [11].

Кроссплатформенный движок может надстраиваться над нативными компонентами интерфейса и инкапсулировать их, но в таком случае возникает сильная связь с особенностями платформы и контроль пове-

[2]Сайт посвящённый Material Design. URL: https://material.io/ (дата обращения: 01.12.2021).

дения приложения будет затрудняться. SLE не использует стандартные компоненты операционных систем, а предоставляет возможность рендеринга собственных виджетов на основе компонентной системы [12]. Обычно фреймворки предоставляют библиотеки готовых виджетов для разработки пользовательского интерфейса. В целях сохранения архитектурной гибкости, производительности и абстрагирования от определённого визуального стиля SLE предлагает использовать компонентную систему для конструирования виджетов самим разработчиком.

Также при разработке кроссплатформенного движка следует учитывать то, что на сегодняшний день различные операционные системы имеют различные API рендеринга, в случае Android — это OpenGL ES [13] и Vulkan [14], а в случае iOS — Metal [15]. Для абстрагирования от вопросов рендеринга и вычислений векторной графики движок SLE использует библиотеку Skia[3], применяемую в браузерном движке Chromium и кроссплатформенном фреймворке Flutter [4].

Важным отличием SLE от Flutter является то, что минимальной единицей интерфейса является компонент, а не виджет. Также Flutter использует язык Dart, а SLE использует C++, что даёт большую эффективность, так как с ростом приложения увеличивается объём кода и производительность начинает резко снижаться, а язык C++ позволяет избежать деградации производительности в долгосрочной перспективе, поскольку C++ является компилируемым языком и использует все возможности оптимизации кода во время компиляции в отличие от JIT-компилируемых и интерпретируемых языков. Покажем эффективность использования памяти Flutter и SLE на примере задач из проекта Big Coloring Book, большое количество визуальных данных требует чисел с плавающей точкой, но не требует слишком большой точности, поэтому для их хранения можно использовать тип float одинарной точности, которого нет в Dart. Уже на этом моменте SLE получает выигрыш в использовании памяти в два раза.

Также Dart имеет более сложную модель памяти и объектов, что не является Zero Cost Abstraction [16]. Компилятор и библиотеки C++ же стремятся изначально к максимальной оптимизации и реализации с учётом принципа «платим только за то, что используем».

SLE позиционируется как движок для написания высокопроизводительных приложений, поэтому для написание всей логики элементов приложения используется современный C++. Данный язык изначаль-

[3]Сайт библиотеки Skia. URL: https://skia.org/ (дата обращения: 01.12.2021).

но создавался для написания высокопроизводительного и безопасного кода при правильном его использовании [17].

С точки зрения SLE, любая логика, реализуемая пользователем, представима в виде компонента и может быть использована повторно. Например, прокручиваемые списки не содержат логики прокрутки, она вынесена в компонент Scroll, и этот же компонент может быть использован для прокрутки частей страницы с большими блоками элементов.

Для присоединения компонент используется абстрактный объект-контейнер, ничего о своих компонентах не знающий. Эти объекты соединяются в деревья. Одно дерево представляет собой одну форму пользовательского интерфейса. Рендеринг происходит от корня интерфейсов, а обработка событий интерфейса, например нажатие на экран, происходит от самого правого листа дерева (последний лист при поиске в глубину).

Так как абстрактные объекты ничего о своих компонентах не знают, они не могут вызывать их функции, для передачи событий в их обработчики компоненты реализуют абстрактные интерфейсы, и объект-контейнер уже может проверить, соответствует ли компонент из списка этому интерфейсу. Если компонент проходит проверку на соответствие, то обработка события передаётся в реализацию функции-обработчика в данном компоненте.

2. Инициализация движка и жизненный цикл приложения Android

Приложения на каждой платформе имеют свой собственный жизненный цикл[4], не похожий друг на друга. Жизненный цикл Android реализуется классами-наследниками Activity[5], различные экраны представляются с помощью различных Activity.

Использование движка предполагает абстрагирование от нативного цикла исполнения приложения, движок должен предоставлять свой жизненный цикл, общий для всех платформ, а за общим циклом уже скрывается нативный. Инициализацию движка и контроль его жизнен-

[4]Жизненный цикл процессов и приложений в Android. URL: https://developer.android.com/guide/components/activities/process-lifecycle (дата обращения: 01.12.2021).

[5]Жизненный цикл Activity в Android. URL: https://developer.android.com/guide/components/activities/activity-lifecycle (дата обращения: 01.12.2021).

ного цикла приходится интегрировать внутрь нативного жизненного цикла приложения.

Для начала рассмотрим, как происходит инициализация движка в Android с использованием языка программирования Java. Код в листинге 1 приведён в упрощённом варианте, опускающем излишние подробности, связанные с проектом Big Coloring Book.

Листинг 1. Фрагмент кода инициализации движка в Android

```
static {
  System.loadLibrary("native-lib");
}
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  this.requestWindowFeature(Window.FEATURE_NO_TITLE);
  getSupportActionBar().hide();
  ClearFiles();
  setContentView(R.layout.activity_main);
  AdView adView = (AdView)findViewById(R.id.adView);
  ...
  startSLEngine(keyboardManager, billingManager, path,
    nativeHelper, socialManager, admobManager,
    firebaseHelper);
  ...
}
```

Вне тела функции происходит загрузка динамической библиотеки с логикой приложения на C++. Основная инициализация движка происходит в функции onCreate [9; 10]. В начале функции идёт основной код, инициализирующий единственное Activity приложения. Далее создаются объекты классов, использующиеся для доступа к нативным функциям Android. Последней вызывается функция `startSLEngine`. Эта функция выполняет основную инициализацию классов движка, и реализована она в C++. Рассмотрим её объявление в Java в листинге 2.

Листинг 2. Объявление функции инициализации движка в Java

```
public native void startSLEngine(Object keyboardManager,
  Object billingManager, String appPath,  Object
     nativeHelper,
  Object playGamesPlugin, Object admobPlugin,
  Object firebaseHelperPlugin);
```

Отметим ключевое слово `native`, оно указывает на то, что эта функция реализована где-то в динамической библиотеке, в следующем разделе будет подробно рассмотрена, как эта функция определена и реализована в C++.

3. Межъязыковое взаимодействие в Java

Следующая проблема, которая возникает при разработке подобного движка — это межъязыковое взаимодействие. В случае с Android C++ и Java код не могут взаимодействовать напрямую никаким образом, так как Java код выполняется виртуальной машиной Java [18], а C++ код — операционной системой. Java библиотеки не являются динамическими библиотеками (*.so для Linux/Unix, *.dll для Windows), и их нельзя вызывать из C++. C++ код можно вызывать из Java кода только при правильном именовании функций, соответствующем документации Oracle по Java Native Interface (JNI). Вызов Java функций аналогично осуществляется с использованием JNI, но ещё более затруднён тем, что в C++ надо сохранять и освобождать глобальные ссылки вручную. Получение доступа к методам осуществляется через сложно читаемую сигнатуру, и в итоге довольно легко допустить утечки памяти, плюс такой код сам по себе довольно трудно отлаживается [7].

Рассмотрим определение функции `startSLEngine` в C++.

Листинг 3. Объявление функции инициализации движка в C++

```
extern "C" JNIEXPORT void JNICALL
Java_sadlion_games_bigcoloringbook_
                            MainActivity_startSLEngine(
    JNIEnv* env,
    jobject obj,
    jobject keyboardManager,
    jobject billingManager,
    jstring appPath,
    jobject nativeHelper,
    jobject playGamesPlugin,
    jobject admobPlugin,
    jobject firebaseHelperPlugin) {
    const char *pchars = env->GetStringUTFChars(appPath,
        NULL);
    string rootPath = pchars;
    env->ReleaseStringUTFChars(appPath, pchars);

    jclass activityClass = env->GetObjectClass(obj);
```

```
    jmethodID method = env->GetMethodID(
        activityClass,
        "getAssetManager",
        "()Ljava/lang/Object;");
    jobject assetManagerObj = env->CallObjectMethod(obj,
        method);
    AAssetManager *aassetManager =
       AAssetManager_fromJava(
        env,
        assetManagerObj);

    engine = new EngineContext(
        env, aassetManager,
        rootPath, keyboardManager,
        billingManager, nativeHelper);
    app = new BigColoringBookApp(engine);
    ...
    app->InitApp(rootPath);
}
```

Особый акцент следует сделать на именовании функции. К имени, объявленном на стороне Java, здесь добавлен префикс с именем пакета, в котором функция будет вызываться. Такое именование функций позволяет инкапсулировать C++ функции в определённых Java-классах. JNICALL разворачивается в ключевые слова, зависящие от операционной системы, помогающие выполнять обнаружение символа функции в файлах *.so/*.dll.

Вначале функции происходит получение объектов, нужных в Android для работы на стороне C++. Через JNI получается класс объекта, находится нужный метод объекта и затем он вызывается [6], таким образом, движок получает менеджер ресурсов операционной системы. Разработчик же использует уже абстракции движка, скрывающие реализацию взаимодействия с менеджером ресурсов. Далее происходит инициализация движка, плагинов и самого приложения. С помощью плагинов, можно расширять функционал движка по взаимодействию с различными сервисами, например сервис мобильной рекламы Admob.

Рассмотрим более подробно объявление, получение и вызов методов через JNI.

```
jmethodID jShowGooglePlay;
```

`jmethodID` является типом, который хранит в себе данные о необходимом методе, для создании переменной этого типа необходимо выполнить следующий код:

```
jShowGooglePlay = env->GetMethodID(
  jNativeHelperClass,
  "ShowGooglePlay",
  "(Ljava/lang/String;)V");
```

Рассмотрим эту строку более подробно. `jNativeHelperClass` указывает класс, в котором находится метод. Получить класс объекта и сохранить его в глобальной ссылке (это не позволит сборщику мусора удалить объект).

```
  jclass objClass = env->GetObjectClass(jnativeHelper);
  this->jNativeHelperClass =
    (jclass)env->NewGlobalRef((jobject)objClass);
```

`jnativeHelper` представляет объект, созданный в onCreate в Java.

Далее в JNI передаётся строка `"ShowGooglePlay"`. Она сообщает системе имя искомого метода.

Последняя строка `"(Ljava/lang/String;)V"` является сигнатурой искомого метода. Она имеет довольно сложный вид и состоит фактически из двух частей. Первая часть `(Ljava/lang/String;)` сообщает, какие аргументы принимает метод, вторая часть `V` — что возвращает искомый метод. Подробно с сигнатурами можно ознакомиться в документации по JNI [6].

Рассмотрим теперь вызов описанной функции:

Листинг 4. Вызов Java функции из C++

```
NativeHelper::ShowGooglePlay(string id)
{
#ifdef ANDROID_ENGINE
    JNIEnv *env;
    int rc = jvm->GetEnv((void **)&env, JNI_VERSION_1_6)
       ;
    bool attached = false;
    if (rc == JNI_EDETACHED) {
        throw "NativeHelper::ShowGooglePlay: thread is
           not attached";
    } else {
        attached = true;
    }
```

```
    if (attached) {
        jstring jid = env->NewStringUTF(id.data());
        env->CallVoidMethod(jnativeHelper,
           jShowGooglePlay, jid);
        env->DeleteLocalRef(jid);
    }
#endif
}
```

Сначала функция из JVM получает переменную среды и затем проверяет, что поток присоединён к виртуальной машине, если поток неприсоединён, то продолжать не имеет смысла. Далее переменная переводится из объекта C++ в Java-объект, после этого происходит вызов функции и стирается ссылка на использованный объект.

4. Инициализация движка и жизненный цикл приложения iOS

В случае iOS код жизненного цикла разделён между несколькими классами-наследниками: `UIView` для экранов, `UIViewController` для управления экранами, `UIApplicationDelegate` для логики и обработки событий [11]. Таким образом инициализация всего приложения разделена между несколькими классами, в которых она происходит. Рассмотрим примеры кода инициализации.

Листинг 5. Инициализация UIView

```
- (id)initWithFrame:(CGRect)frame {
    nfingers = 0;
    clickX = clickY = 0.0f;
    mayBeClick = mayBeSwipe = false;
    swipeDx = swipeDy = 0.0f;
    lastX = lastY = 0.0f;
    prevScale = 1.0f;
    touchChron = make_shared<Chronograph>();
    ...
    if ((self = [super initWithFrame:frame])) {
        self.textStore = [NSMutableString string];
        [self.textStore appendString:@""];
    }
    return self;
}
```

В данном случае приведена функция инициализации для класса-наследника `UIView`. В ней инициализируются связанные с экраном системы, например: обработка жестов, обработка виртуальной клавиатуры. Рассмотрим далее инициализацию класса наследника `UIViewController`.

Листинг 6. Инициализация UIViewController

```
- (void)viewDidLoad {
    [super viewDidLoad];
    _supportedInterfaceOrientations =
       UIInterfaceOrientationPortrait;
        mView = (MTKView*)[KeyInputView new];
    ...
    AppDelegate *appDelegate = (AppDelegate*)[[
       UIApplication sharedApplication] delegate];
        mView.enableSetNeedsDisplay = NO;
        mView.device = MTLCreateSystemDefaultDevice();

    [appDelegate initWithMTKView: mView];
        mView.delegate = (id<MTKViewDelegate>)
           appDelegate;
        mView.clearColor = MTLClearColorMake(0.0, 1.0,
           0.0, 1.0);
    ...
}
```

Данный код инициализирует системы, управляющие экранами, например ограничение расположения экранов с помощью «безопасной зоны», определяющей зону устройства, где только экран и нет никаких других элементов вроде «челок». Также в данной функции создаётся основной экран приложения с поддержкой системы рендеринга Metal [15]. Последняя часть инициализации расположена в классе-наследнике `UIApplicationDelegate`. Далее приведён последний фрагмент инициализации.

Листинг 7. Инициализация UIApplicationDelegate

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)
       launchOptions {
    productIds = [NSMutableArray new];
    products = [NSMutableDictionary<NSString*, SKProduct
       *> new];
```

```
        [[SKPaymentQueue defaultQueue]
           addTransactionObserver: self];

    _currentCredential = nil;

    [UNUserNotificationCenter currentNotificationCenter
       ].delegate = self;
    UNAuthorizationOptions authOptions =
       UNAuthorizationOptionAlert |
       UNAuthorizationOptionSound |
       UNAuthorizationOptionAlert;
        [[UNUserNotificationCenter
           currentNotificationCenter]
           requestAuthorizationWithOptions:authOptions
           completionHandler:^(BOOL granted, NSError *
           _Nullable error) {
            if (error) {
                NSLog(@"Fail to register for push");
            }
        }];

    [application registerForRemoteNotifications];
    return YES;
}
```

В данной функции происходит инициализация систем, связанных с уведомлениями и обработкой нативных событий, например совершение платежей в магазине приложения или обработка пуш-уведомлений.

5. Межъязыковое взаимодействие в Objective-C

В случае с iOS проблема межязыкового взаимодействия ощущается не так сильно, если использовать Objective-C++ для разработки нативной части так, как классы и функции C++ можно использовать сразу в коде на Objective-C++, так и, наоборот, создавая своеобразный микс из кода [11]. Следующий отрывок наглядно показывает, как выглядит взаимодействие двух указанных языков.

Листинг 8. Межъязыковое взаимодействие в Objective-C

```
[cropController
    dismissViewControllerAnimated:TRUE completion:^{
        NSData* pngData = UIImagePNGRepresentation(image
           );
```

```
        auto nativeHelper = app->GetEngineContext()->
           GetNativeHelper();
        nativeHelper->
           DoPickupImageFromGalleryAndCropCallback(
            (unsigned char*)pngData.bytes,
            pngData.length);
    }];
```

6. Платформозависимый код

Последняя трудность, связанная с написанием кода при разработке движка, — написание платформозависимого кода. Абстракция нативного кода, с которой работает пользователь, одновременно содержит в себе код для всех платформ, разделённый условной компиляцией, он может выглядеть немного пугающе, и исходные файлы становятся значительно больше. Пример кроссплатформенного кода можно увидеть ниже.

Листинг 9. Платформозависимый код

```
int
NativeHelper::GetTimezone()
{
#ifdef ANDROID_ENGINE
    JNIEnv *env;
    int rc = jvm->GetEnv((void **) &env, JNI_VERSION_1_6
       );
    if (rc == JNI_EDETACHED) {
        throw "NativeHelper::GetTimezone: thread is not
           attached";
    }
    return env->CallIntMethod(jnativeHelper,
       jGetTimezone);
#elif IOS_ENGINE
    return (int)[[NSTimeZone localTimeZone]
       secondsFromGMT];
#else
    throw "not implemented";
    return -100;
#endif
}
```

7. Заключение

В результате работы SLE получил поддержку двух платформ: Android и iOS. Опыт работы с прочими кроссплатформенными фреймворками показал, что высокоуровневые фреймворки хороши при прототипировании приложения, но в долгосрочной перспективе использование SLE решает ряд проблем:

1. Архитектурное устройство на основе компонентной системы позволяет создавать приложение с дизайном любой сложности и не зависеть от компонентов операционной системы или компонентов, предоставляемых кроссплатформенным фреймворком.

2. Абстрагирование логики приложения от платформенных особенностей позволяет разрабатывать приложения сразу для нескольких платформ, имея при этом только одну кодовую базу.

3. Использование C++ позволяет быстро расширять взаимодействие с нативным функционалом платформы, сохраняя при этом производительность приложения.

4. Опыт разработки SLE показал, что для достижения высокого качества и частоты кадров как на современных, так и устаревших устройствах для отрисовки интерфейсов необходимо использовать GPU, а для работы с векторной графикой — библиотеку Skia.

5. Использование стандартных средств разработки: Android Studio и XCode, значительно облегчает отладку взаимодействия нативного кода и SLE.

Список литературы

1. Bosnic S., Papp I. The development of hybrid mobile applications with Apache Cordova // *24th Telecommunications Forum.* 2016. Pp. 1–4.

2. Tomozei C. Assessment of the evolution in quality for Xamarin Android Multimedia Applications // *19th International Conference on Informatics in Economy. Education, Research and Business Technologies.* 2020. Pp. 47–52.

3. Мельников В. А. Процесс разработки движка для 2D игр и интерфейсов Sad Lion Engine // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика.* 2019. Вып. 4 (33). С. 21–37.

4. Fisz K., Kopniak P., Galan D. A multi-criteria comparison of mobile applications built with the use of Android and Flutter Software Development Kits // *Journal of Computer Sciences Institute*. Vol. 19. 2021. Pp. 107−113.

5. Dabit N. React Native in action. Shelter Island: Manning Publishing. 2019. 320 p.

6. Java Native Interface Specification [Online] //textitOracle. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html (Accessed: 22.10.2020).

7. Rago S., Stevens W. Advanced programming in the UNIX environment, 3rd ed. Edition. Upper Saddle River, NJ, Boston, Indianopolis, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City: Addison-Wesley, 2013. 1032 p.

8. Corbet J., Kroah-Hartman G., McKellar J., Rubini A. Linux device drivers, 4th ed. Beijing, Cambridge, Farnham, K?ln, Sebastopol, Taipei, Tokyo, 2017. 600 p.

9. Thornsby J. Android UI design. Birmingham: Packt Publishing, 2016. 356 p.

10. Anugerah M. A., Sekar G. S. Designing Android User Interface for University Mobile Library // *International Conference on Computing, Engineering, and Design (ICCED)*. 2021. Pp. 224−229.

11. Neuburg M. Programming iOS 13: Dive Deep into Views, View Controllers, and Frameworks. Sebastopol: O'Reilly. New York: Oracle, 2020. 1208 p.

12. Nystrom R. Game programming patterns, San Bernardino: Genever Benning, 2018. 345 p.

13. Ginsburg D., Purnomo B. OpenGL ES 3.0 Programming Guide 2nd Edition. Upper Saddle River, NJ, Boston, Indianopolis, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City: Addison-Wesley, 2014. 560 p.

14. Sellers G. Vulkan Programming Guide. The Official Guide to Learning. Vulkan, Boston, Columbus, Indianapolis, New York, San Francisco, Amsterdam, Cape Town Dubai, London, Madrid, Milan, Munich, Paris, Montreal, Toronto, Delhi, Mexico City San Paulo, Sydney, Hong Kong, Seoul, Singapore, Taipei, Tokyo: Addison-Wesley, 2017. 480 p.

15. Clayton J. Metal programming guide. Addison-Wesley, 2018. 352 p.

16. Stroustrup B. The C++ programming languege 4th edition. Uppder Saddle River, Boston, Indianopolis, San Francisco, New York, Toronto, Montral, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico city: Addison-Wesley, 2013. 1345 p.

17. Stroustrup B. Programming: Principles and Practice using C++. 2nd Edition. New Jearsey: Pearson Education, 2015. 1312 p.

18. Shieldt H. Java: The Complete Reference, Eleventh Edition 11th Edition. 2019. 1248 p.

Summary

Yermolenko A. V., Melnikov V. A. Solving the problem of abstraction from platform-specific code for iOS and Android applications using the example of SadLion Engine

The paper examines existing solutions for cross-platform mobile development, compares their features, advantages and disadvantages. It describes the solution to various problems arising in the development of your own cross-platform engine for development for iOS and Android. The construction of a system for displaying a visual interface on a user screen using a GPU is considered. The architectural solutions used to write high-performance logic of application behavior in the C ++ programming language are described. The life cycles of applications for the iOS and Android platforms are considered and a way to abstract from the native life cycle is proposed to generalize the application code on both platforms.

The implementation of interlanguage interaction between Java and C ++ using JNI on the Android platform and Objective-C and C ++ is described, architectural solutions are given for building an abstraction layer that hides such low-level interactions in the engine core.

Keywords: cross-platform development, C ++, Android, iOS.

References

1. Bosnic S., Papp I. The development of hybrid mobile applications with Apache Cordova. *24th Telecommunications Forum.* 2016. Pp. 1−4.

2. Tomozei C. Assessment of the evolution in quality for Xamarin Android Multimedia Applications. *19th International Conference on Informatics in Economy. Education, Research and Business Technologies.* 2020. Pp. 47−52.

3. Melnikov V. A. Development Process of game engine core for 2D games and interfaces Sad Lion Engine. *Vestnik Syktyvkarskogo universiteta. Ser. 1: Matematika. Mexanika. Informatika* [Bulletin of Syktyvkar University. Series 1: Mathematics. Mechanics. Informatics], 2019, 4 (33). Pp. 21–37.

4. Fisz K., Kopniak P., Galan D. A multi-criteria comparison of mobile applications built with the use of Android and Flutter Software Development Kits. *Journal of Computer Sciences Institute.* Vol. 19. 2021. Pp. 107−113.

5. Dabit N. *React Native in action.* Shelter Island: Manning Publishing. 2019. 320 p.

6. Java Native Interface Specification [Online]. *Oracle.* Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html (Accessed: 22.10.2020).

7. Rago S., Stevens W. *Advanced programming in the UNIX environment, 3rd ed.* Upper Saddle River, NJ, Boston, Indianopolis, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City: Addison-Wesley, 2013. 1032 p.

8. Corbet J., Kroah-Hartman G., McKellar J., Rubini A. *Linux device drivers, 4th ed.* Beijing, Cambridge, Farnham, Koln, Sebastopol, Taipei, Tokyo. 2017. 600 p.

9. Thornsby J. *Android UI design.* Birmingham: Packt Publishing. 2016. 356 p.

10. Anugerah M. A., Sekar G. S. Designing Android User Interface for University Mobile Library. *International Conference on Computing, Engineering, and Design (ICCED).* 2021. Pp. 224−229.

11. Neuburg M. *Programming iOS 13: Dive Deep into Views, View Controllers, and Frameworks.* Sebastopol: O'Reilly. 2020. 1208 p. New York: Oracle.

12. Nystrom R. *Game programming patterns.* San Bernardino: Genever Benning, 2018. 345 p.

13. Ginsburg D., Purnomo B. *OpenGL ES 3.0 Programming Guide 2nd Edition.* Upper Saddle River, NJ, Boston, Indianopolis, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City: Addison-Wesley, 2014. 560 p.

14. Sellers G. *Vulkan Programming Guide. The Official Guide to Learning Vulkan.* Boston, Columbus, Indianapolis, New York, San Francisco, Amsterdam, Cape Town Dubai, London, Madrid, Milan, Munich, Paris, Montreal, Toronto, Delhi, Mexico City San Paulo, Sydney, Hong Kong, Seoul, Singapore, Taipei, Tokyo: Addison-Wesley, 2017. 480 p.

15. Clayton J. *Metal programming guide.* Addison-Wesley. 2018. 352 p.

16. Stroustrup B. *The C++ programming languege 4th edition.* Uppder Saddle River, Boston, Indianopolis, San Francisco, New York, Toronto, Montral, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico city: Addison-Wesley. 2013. 1345 p.

17. Stroustrup B. *Programming: Principles and Practice using C++. 2nd Edition.* New Jearsey: Pearson Education. 2015. 1312 p.

18. Shieldt H. *Java: The Complete Reference, Eleventh Edition 11th Edition.* 2019. 1248 p.

Для цитирования: Ермоленко А. В., Мельников В. А. Решение проблемы абстрагирования от платформозависимого кода // *Вестник Сыктывкарского университета. Сер. 1: Математика. Механика. Информатика.* 2021. Вып. 4 (41). С. 50–69. DOI: 10.34130/1992-2752_2021_4_50

For citation: Yermolenko A. V., Melnikov V. A. Solving the problem of abstraction from platform-specific code for iOS and Android applications using the example of SadLion Engine. *Bulletin of Syktyvkar University, Series 1: Mathematics. Mechanics. Informatics*, 2021, No. 4 (41), pp. 50–69. DOI: 10.34130/1992-2752_2021_4_50

Сыктывкарский государственный университет им. Питирима Сорокина

Поступила 29.11.2021