Министерство образования и науки Российской Федерации
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)
ФАКУЛЬТЕТ РАДИОТЕХНИКИ И КИБЕРНЕТИКИ
КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ

РУДОЙ АРТЁМ ИГОРЕВИЧ

ВИРТУАЛИЗАЦИЯ ГРАФИЧЕСКОЙ ПОДСИСТЕМЫ ANDROID НА УРОВНЕ OPENGL

Выпускная квалификационная работа бакалавра

Направление подготовки 010900 «Прикладные математика и физика»

Научный руководитель _____ Симановский А. Ю.

Студент _____ Рудой А. И.

г. Долгопрудный
2016

Аннотация

В данной выпускной квалификационной работе представлен подход к виртуализации OpenGL ES и EGL в Android ОС.

Описаны различные способы передачи данных между гостевой и хостовой ОС. Описан выбор наиболее подходящего для конкретной задачи способа передачи данных. Описан способ трансляции OpenGL ES вызовов в DirectX.

Описан способ подмены OpenGL ES и EGL библиотеки. А так же на основе вышеперечисленных технологий, описан способ реализации библиотеки, которая виртуализирует часть OpenGL ES и EGL API.

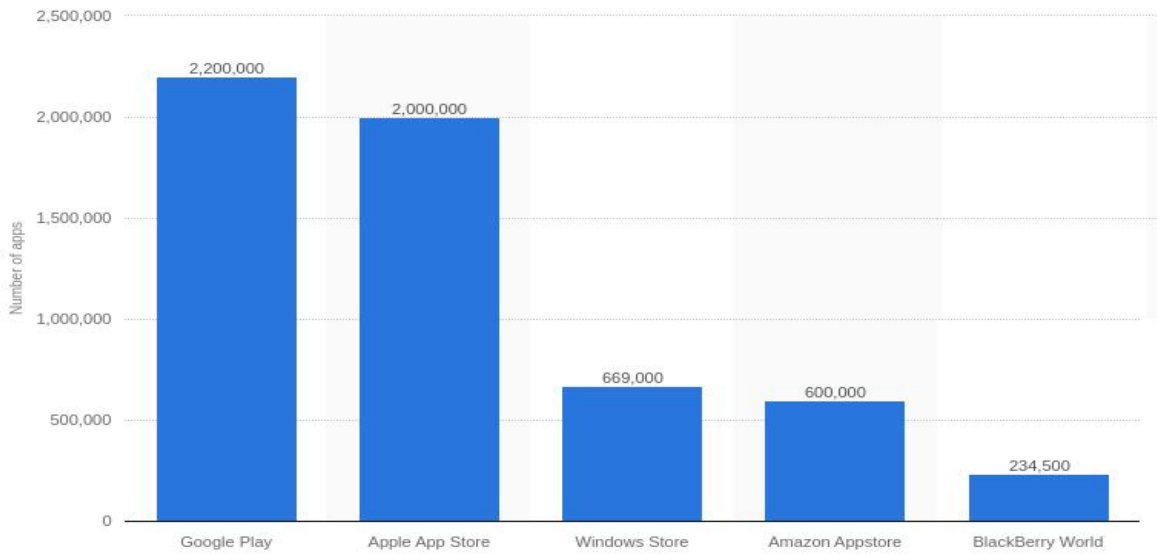
Оглавление

Глава 1

Введение

1.1 Предпосылки

На момент написания этой дипломной работы доля портативных устройств на базе операционной систем Windows 8.1 в сравнении с устройствами на базе Android и iOS была крайне мала. На данный момент в Microsoft Store’е выложено около 700 000 приложений. Для сравнения в Google Play их количество составляет около 2 200 000 , а в Apple Store их порядка 2 000 000. Более подробные цифры можно увидеть на диаграмме:



Предполагалось, что если удастся запускать приложения из Google Play на планшетах на базе операционной системы Windows 8.1, то это поможет увеличить их долю на рынке.

Для осуществления этой идеи необходимо на планшетах с хостовой операционной системой Windows 8.1 запускать виртуальную машину с гостевой операционной системой Android. Однако, для комфортной работы в этой экосистеме этого будет недостаточно, поскольку существующие открытые реализации Android’а никак не оптимизированы для запуска в качестве гостевой операционной системы: в лучшем случае присутствующие устройства(например, видео-карта) - эмулируются, а в худшем, просто отсутствуют. В итоге, в случае эмуляции, все действия будут крайне медленны по сравнению с работой на настоящих устройствах, что , очевидно, отпугнет пользователей от работы на планшетах с такой технологией. Поэтому нужно каким-то образом решить эту проблему. Однако, в общем случае ее решить невозможно: каждое устройство требует индивидуальную программную поддержку аппаратных реализаций разных производителей. Поэтому целью данной дипломной работы является изучение способов виртуализации графической подсистемы Android’a.

1.2 Постановка задачи

Сам процесс виртуализации для графической подсистемы Android ОС подразумевает, что действия, которые изначально исполнялись на эмулируемой видео-карте перенаправляются на хостовую ОС (Windows 8.1) , где выполняются там на полноценной видео-карте, а результат этих вызовов передается обратно в гостевую систему, где уже дальше обрабатывается гостевой системой.

Для того, чтобы приложение использовало ресурсы видео-карты для рендеринга графики, программист должен эту программу писать, используя специальные интерфейсы программирования приложений (далее API). На данный момент, существуют два таких независимых API: DirectX и OpenGL. Несмотря на то, что они решают похожие задачи, эти API разработаны разными компаниями, поэтому они не являются бинарно совместимыми.

В операционной системе Android используется OpenGL ES API для рендеринга графики. В открытых реализациях Android ОС вызовы к OpenGL ES API исполняются на эмулируемой видео-карте, что, как писалось ранее, существенно уменьшают скорость выполнения программ, кроме того, в открытых реализациях Android имеется только OpenGL ES 1.0, а большинство программ уже используют OpenGL ES 2.0 и выше, что не позволяет большинству программ даже запускаться. Про OpenGL ES подробнее будет написано в следующей главе. Исходная задача усложняется еще тем, что Microsoft Windows системы используют DirectX в качестве API для рендеринга изображений, а как писалось ранее, OpenGL ES и DirectX не совместимы друг с другом.

Итак, чтобы виртуализировать графическую подсистему Android на уровне OpenGL ES необходимо решить три практически независимые друг от друга задачи:

1. реализовать канал передачи данных между гостевой и хостовой ОС
2. найти способ конвертации OpenGL ES в DirectX
3. подменить гостевую OpenGL ES библиотеку, которая, используя канал передачи данных и OpenGL ES транслятор, перенаправляет вызовы из гостя в хостовую ОС, там их исполняет, а результат обратно возвращает в гостевую ОС

1.3 Программное окружение

Поскольку во время исследований программное окружение должно быть близко к постановленному в задаче, а именно, к планшетам на x86 платформе с операционной системой Windows 8, то все исследования проводились на 32-битной версии Windows 8.1. В качестве виртуальной машины использовался Oracle VM VirtualBox. VirtualBox был выбран неслучайно: исходный код продукта открыт, что позволяет его модифицировать, а так же при встрече плоходокументированных моментов изучать и находить ответы на возникшие вопросы непосредственно в исходном коде программы[1]. Android был взят версии API 19[6]. На момент написания диплома количество устройств на этой версии ОС было порядка 31.6%, что является самым высоким показателем среди всех других версий Android.

Глава 2

Введение в OpenGL ES и EGL

OpenGL ES представляет собой API для программирования 3D графики, ориентированный на портативные и встраиваемые устройства, такие как сотовые телефоны, карманные персональные компьютеры (PDA), игровые консоли, транспортные средства и авионики. OpenGL ES является одним из набора API-интерфейсов, созданных Khronos Group. Khronos Group, основанная в январе 2000 года, является промышленный консорциумом, целью которого является выработка открытых API в области создания и воспроизведения динамической графики и звука на широком спектре платформ и устройств, с поддержкой аппаратного ускорения[3].

На данный момент в настольном мире есть два стандартных 3D API, DirectX и OpenGL. DirectX является стандартом 3D API для любой системы запущенной на Mocrosoft Windows и используется большинством 3D игр на этой платформе. OpenGL является кросс-платформенным стандартом 3D API для настольных систем под управлением Linux, различных версий UNIX, Mac OS X, и Microsoft Windows.

В свою очередь, на данный момент существует несколько спецификаций OpenGL ES, которые были выпущены Khronos Group: OpenGL ES 1.0, ES 1.1, ES 2.0 и 3.0. В OpenGL ES 2.0 и выше используется программируемый графический конвейер. Это означает, что вся ответственность за положение камер, освещение, эффекты лежит полностью на разработчиках. Делается все это при помощи программирования шейдеров . В свою очередь, шейдеры - это программы, написанные на языке по синтаксису похожем на язык C: OpenGL ES Shader Language (GLSL ES или ESSL). Сначала разработчик пишет программу на этом языке, далее при помощи функций OpenGL ES она отправляется в ядро OpenGL, там компилируются и позже исполняется.

Шейдеры бывают двух типов: вершинные и фрагментные. Вершинные шейдеры обычно используются для традиционных векторных операций, таких как: изменения положения координат вершин, расчет влияния освещения на цвет вершины, генерация или трансформация координат текстур. В вершинные шейдеры передаются координаты вершин. После отработки вершинных шейдеров наступает процесс растеризации: OpenGL ES собирает всю информацию о 3D объектах (координаты вершин, положения камер) и создает 2D изображение. Далее фрагментные шейдеры обрабатывают каждую видимую часть конечного изображения, в результате высчитывая цвет каждого пикселя изображения. Тогда как версии спецификаций 1.0 и 1.1 описывают фиксированный графический конвейер. Это означает, что разработчикам дается фиксированный набор функций для управления камерой, освещением, эффектами, материалами.

OpenGL ES 2.0 не обладает обратной совместимостью с OpenGL ES 1.x. Эта версия спецификации не поддерживает функции фиксированного конвейера, представленного в OpenGL ES 1.x.. Так, программируемые вершинные шейдеры OpenGL ES 2.0 заменяют фиксированные функции по

работе над вершинами, реализованные в OpenGL ES 1.x. Похожим образом фрагментные шейдеры заменяет фиксированный набор функций, используемый для обработки текстур, представленный в OpenGL ES 1.x[9].

Однако, команды OpenGL ES требуют контекст для рендеринга и поверхности для рисования. Контекст для рендеринга хранит текущее состояние OpenGL ES. А поверхность для рисования - это область куда будут рисоваться примитивы. Поверхность для рисования определяет различные типы буферов, которые будут использоваться в процессе рендеринга: цветовой буффер (color buffer,) , буффер глубины(depth buffer) и буфер шаблона(stencil buffer).

OpenGL ES API не заботится о создании контекста для рендеринга, а так же каким образом этот контекст присоединяется к оконной системе операционной системы. За это ответственно другое API - EGL. EGL - это интерфейс между OpenGL ES API и оконной системой операционной системы. Инными словами, OpenGL ES ответственен за то, что изображать, а EGL - за то, как это изображение разместить на экране устройства. В функции EGL также входит запрос доступных для отображения изображения дисплеев устройства, создание поверхности для рендеринга. Поверхности для рендеринга бывают двух типов: видимые и теневые. Видимые присоединены к оконной системе операционной системы для отображения изображения на экране устройства, в своё время как теневые используются в качестве буфферов для рендеринга. Так же EGL API ответственен за создание контекста рендеринга[9].

Глава 3

Описание модели

В данной главе будет описано каким образом решалась поставленная задача. Чтобы виртуализировать графическую подсистему Android на уровне OpenGL ES, как писалось ранее, необходимо решить три практически независимые друг от друга задачи: во-первых , необходимо обеспечить канал передачи данных между гостем и хостовой ОС, во-вторых, требуется найти способ конвертации OpenGL ES в DirectX, и в-третьих, нужно на хостовой и гостевой операционной системе подменить гостевую OpenGL ES библиотеку, которая, используя канал передачи данных и OpenGL ES транслятор, перенаправляет вызовы из гостя в хост, там их исполняет, а результат обратно возвращает в гостевую ОС.

3.1 Канал передачи данных

Прежде чем выбирать канал передачи данных, необходимо сформулировать для него дополнительные требования. Во-первых, в рамках поставленной задачи гость и хост - совершенно разные операционные системы: Android и Windows 8.1, поэтому канал передачи данных должен был иметь реализации на этих операционных системах. Во-вторых, в рамках поставленной задачи, канал должен поддерживать возможность передачи данных между различными процессами, а также описывать протокол вызова соответствующих функций на хостовой ОС. В-третьих, канал передачи данных должен обладать высокой пропускной способностью.

В итоге, после рассмотрения существующих решений, выбор был остановлен на сокетах и VirtualBox toolgate’e. Рассмотрим каждый из инструментов подробнее.

3.1.1 Сокеты

Сокеты — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Потенциально сокеты могут выступать в качестве канала передачи данных, поскольку их реализация имеется на требуемых операционных системах.

3.1.2 VirtualBox Host-Guest Communication Manager

В VirtualBox имеется специальный модуль Host-Guest Communication Manager (HGCM) , который позволяет гостевым приложениям или гостевым драйверам вызывать функции хостовой библиотеки. В самом VirtualBox’e Shared Folders, Shared Clipboard и Guest configuration interface используют HGCM в качестве канала передачи данных[1].

HGCM использует VMM виртуальное PCI-устройство для обмена данными между гостем и хостом. Гость всегда является инициатором процесса передачи данных. Запрос формируется в гостевой физической памяти, которая должна быть заблокирована гостем. Физический адрес передается VMM устройству путем 32-битной out edx, eax инструкции. В случае 64-битного гостя, физическая память должна быдь выделена в первых 4 ГБ физической памяти.

Хост разбирает пришедшие от гостя заголовки и данные запроса и передает их в HGCM сервис. В это время гость продолжает выполняться и, как правило, блокируется пока не придет ответ от хостовой ОС.

Когда запрос полностью исполнится HGCM сервисом, VMM устройство устанавливает флаг завершения в заголовок запроса , устанавливает HGCM событие и вызывает IRQ для гостя. Далее HGCM проверяет наличие флага завершения в заголовке запроса и в случае его наличия , запрос считается завершенным.

Описание протокола находится в заголовочном файле VBox/VBoxGuest.h.

Чтобы воспользоваться этим протоколом , необходимо проделать следующие шаги:

1. Реализовать клиентское-гостевое приложение, которое обращается к сервису, а так же сам HGCM-сервис.

2. Во время исполнения клиентского приложения необходимо подключиться к HGCM-сервису.
3. После успешного подключения можно выполнять передачу данных и вызывать хостовый сервис.
4. После передачи данных отключиться от сервиса.

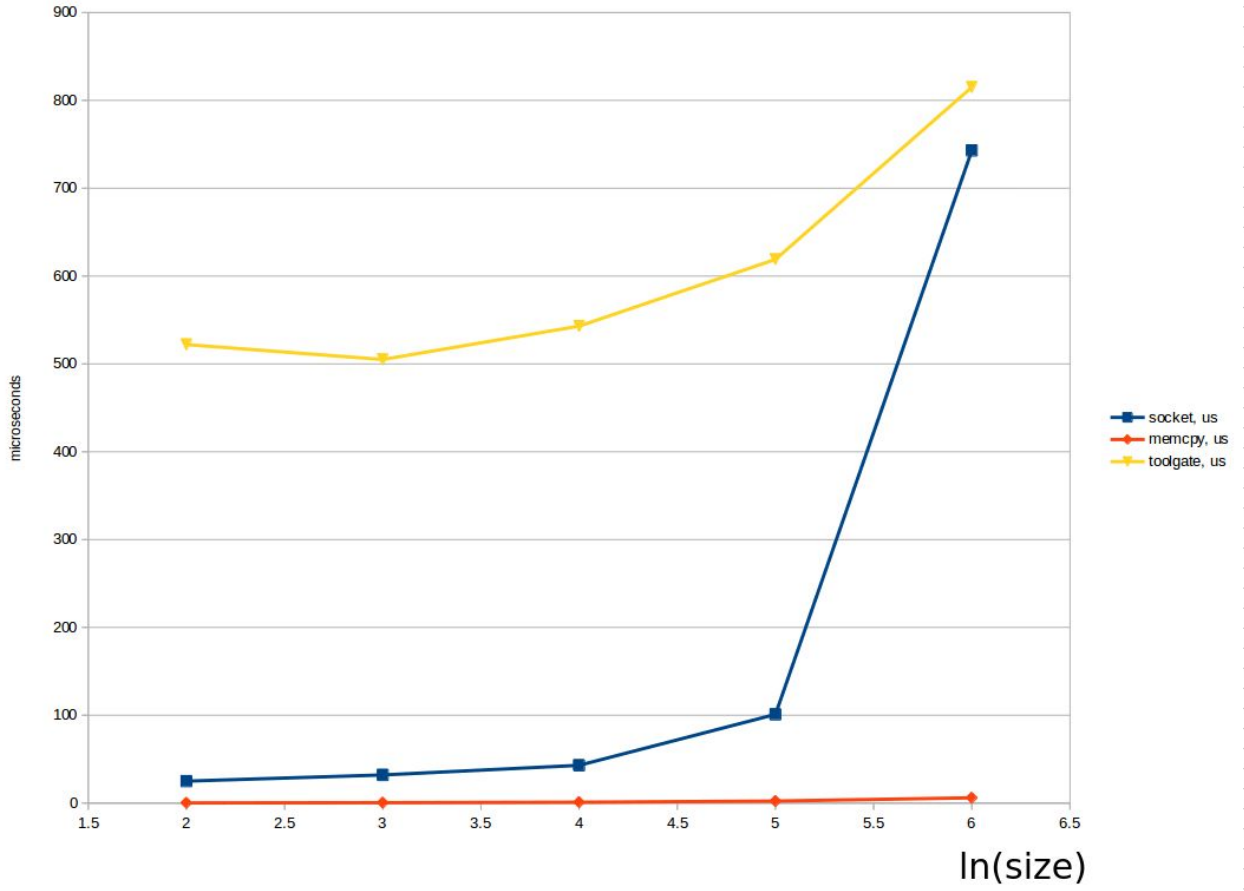
В описании работы протокола HGCM сказано, что в качестве гостевых HGCM клиентов могут выступать как драйвера, так и обычные приложения уровня пользователя. Однако механизм установки соединения и передачи данных отличается для приложений уровня пользователя и уровня ядра. Поскольку в данной работе необходимо было реализовать передачу данных OpenGL ES и EGL , а это библиотеки уровня пользователя, то более подробно установимся на описании реализации передачи данных уровня пользователя.

Приложения гостевой ОС вызывают VirtualBox Guest Additions драйвер, чтобы воспользоваться HGCM интерфейсом. Для того, чтобы определить какого типа вызов: соединение, отсоединение или обращение к функциям HGCM сервиса, в качестве параметра передается константа: VBOXGUEST_IOCTL_HGCM_CONNECT, VBOXGUEST_IOCTL_HGCM_DISCONNECT и VBOXGUEST_IOCTL_HGCM_CALL соответственно. Для подключения к сервису заполняется структура VBoxGuestHGCMConnectInfo, куда передается название хостового сервиса. Результатом успешного подключения к сервису является его числовой идентификатор, который содержится в поле u32ClientID. Далее этот идентификатор передается во всех последующих обращениях к сервису. Аналогично выполняется отключение: заполняется VBoxGuestHGCMDisconnectInfo, а вместо названия сервиса передается его числовой идентификатор, который был получен при подключении к сервису. Далее для того, чтобы осуществить вызов функций хостового сервиса,

необходимо сформировать буфер, в начале которого содержится заполненная структура VBoxGuestHGCMCallInfo, в которой указывается числовой идентификатор подключаемого сервиса, количество передаваемых аргументов, а также номер функции, которая должна выполняться в хостовом сервисе. Для каждого из передаваемых параметров заполняется структура HGCMFunctionParameter, в которой выбирается тип передаваемого параметра(32-битное значение, 64-битное значение, 32-битный гостевой физический адрес, 32-битный гостевой линейный адрес и т.д.), а также размер передаваемых данных и указатель на местонахождение параметра. Далее зовется ioctl с параметром VBOXGUEST_IOCTL_HGCM_CALL и на хостевой ОС вызывается функция , номер которой был передан в соответствующем поле структуры VBoxGuestHGCMCallInfo[7].

3.1.3 Тестирование канала передачи данных

Далее в рамках тестирования этих двух каналов передачи данных были написаны простые программы, целью которых было тестирование скорости передачи данных.



Для наглядности, помимо сокетов и HGCM на графике отображена скорость работы функции, которая просто копирует буфер из одного места в другое (memcpy()).

Как видно из результатов тестирования, скорость передачи данных через сокеты в разы превосходит скорость передачи данных через HGCM. Однако, использование сокетов в качестве канала передачи данных порождает ряд других проблем: для передачи информации о том, какая функция должна быть вызвана на хостовой ОС, необходимо реализовывать на госте и хосте сервисы по функционалу похожие на HGCM, что является нетривиальной задачей и выходит за рамки данной дипломной работы. Поэтому в качестве канала передачи данных был выбран HGCM.

3.2 Трансляция OpenGL ES и EGL в DirectX

Следующий вопрос, который необходимо было решить - это трансляция OpenGL ES вызовов в DirectX. Про OpenGL и DirectX подробно было описано в главе 1.

Поскольку написание своего транслятора выходило за рамки данной дипломной работы, то были рассмотрены уже существующие решения. В качестве такого решения был взят Almost Native Graphics Layer Engine (ANGLE)[5].

ANGLE позволяет пользователям Windows запускать OpenGL ES приложения путем эффективной трансляции OpenGL ES API в DirectX API вызовы. В момент анонса ANGLE в 2010 году основной его целью называлось возможность запускать WebGL приложения в браузерах на Windows без использования OpenGL драйверов.

В свою очередь, WebGL - программная библиотека для языка программирования JavaScript, позволяющая создавать на JavaScript интерактивную 3D-графику, функционирующую в широком спектре совместимых с ней веб-браузеров. За счёт использования низкоуровневых средств поддержки OpenGL, часть кода на WebGL может выполняться непосредственно на видеокартах. WebGL — это контекст элемента canvas HTML, который обеспечивает API 3D графики без использования плагинов. Спецификация версии 1.0 была выпущена 3 марта 2011 года. Проект по созданию библиотеки управляется некоммерческой организацией Khronos Group.

ANGLE используется по умолчанию для ускорения WebGL в Google Chrome и Mozilla Firefox в браузерах для Windows. Кроме того, Chrome

использует ANGLE для всего графического рендеринга, включая ускорение Canvas2D.

В процессе проектирования ANGLE разработчикам удалось решить задачи связанные с различными возможностями и конвенциями DirectX и OpenGL ES. Например, различия в координатной конвенции: в OpenGL используется правосторонняя система координат, а в DirectX - левосторонняя, так же была решена проблема трансляции шейдеров OpenGL ES в их DirectX аналоги и многие другие.

Чтобы проверить скорость работы ANGLE, были взяты результаты тестирования его API на стандартных тестах:

	Desktop GL (fps)	ANGLE (fps)
MapsGL, San Francisco street level	32	33
WebGL Field, “lots” setting	25-48	25-45
Flight of the Navigator	20 минимум	40 минимум
Skin rendering	62	53

Исходя из данных результатов можно сказать, что дополнительная трансляция не дает значительные издержки в производительности, поэтому ANGLE был выбран в качестве транслятора OpenGL ES в DirectX.

3.3 Виртуализация OpenGL ES и EGL

В предыдущих разделах было описано, каким образом был выбран удовлетворяющий критериям данной дипломной работы канал передачи данных,

способ вызовов функций на хостовой системе, а также способ трансляции вызовов OpenGL ES API в DirectX API. В данном разделе будет описано каким образом эти компоненты объединяются, а так же будет более подробно описан способ виртуализации OpenGL ES и EGL.

Суть виртуализации графической подсистемы на уровне OpenGL ES и EGL сводится к подмене стандартной реализации этих библиотек на библиотеки, которые непосредственно будут отвечать за трансляцию вызовов гостевой ОС в соответствующие вызовы хостовой ОС, а так же будут учитывать некоторые особенности запуска Android в качестве гостевой ОС. Однако, чтобы понять каким образом эти библиотеки можно подменить, сначала нужно разобраться в какое место графического стека Android ОС нужно встраиваться. Для этого необходимо понять каким образом данная операционная система взаимодействует с существующей реализацией библиотек. Об подробнее этом будет рассказано в следующем разделе.

3.3.1 Взаимодействие Android ОС с OpenGL ES и EGL

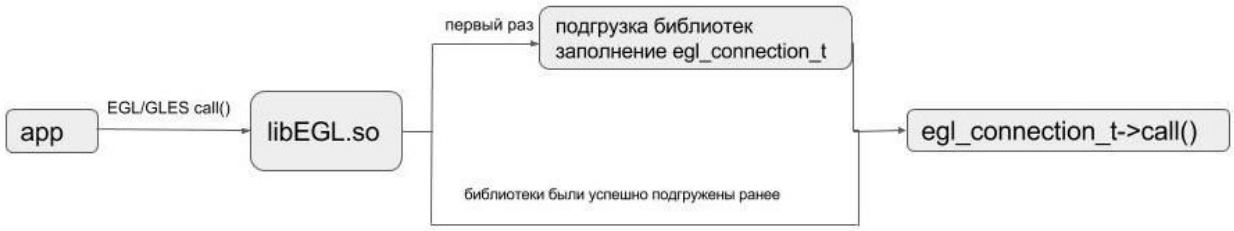
Как писалось ранее, OpenGL ES и EGL предоставляют набор функций, которым могут пользоваться разработчики, а их реализация полностью зависит от производителя GPU[3].

В Android’е загрузка нужной версии библиотек OpenGL ES и EGL API происходит динамически, а сам процесс состоит из двух частей: сначала загружается системный компонент, который, по факту, является уже оберткой

над производителезависиомой версии библиотеки, а потом он уже динамически подгружает нужные OpenGL ES и EGL библиотеки. Остановимся на этом процессе поподробнее.

Как было описано ранее, чтобы воспользоваться OpenGL ES API необходимо провести ряд настроек, используя EGL: выбрать дисплей, на котором будет отображаться отрендеренное изображение, создать поверхности для рисования, создать контекст. Поэтому в системной библиотеке libEGL.so в ряде функций, отвечающих за настройку OpenGL ES , таких как: EGLDisplay eglGetDisplay(EGLNativeDisplayType display), __eglMustCastToProperFunctionPointerType eglGetProcAddress(const char *procname), EGLBoolean eglBindAPI(EGLenum api), EGLenum eglQueryAPI(void), EGLuint64NV eglGetSystemTimeFrequencyNV() и других, вызывается функция EGLBoolean egl_init_drivers(). Именно эта функция отвечает за динамическую подгрузку OpenGL ES и EGL библиотек. Алгоритм выбора нужной библиотеки в Anroid довольно-таки прост: сначала ищутся библиотеки с именем libGLES_*.so(в ней разработчики GPU должны реализовать OpenGL ES и EGL API для конкретного чипа) , которые находятся в папке /vendor/lib/egl/, если таких нет, то по отдельности ищутся библиотеки lib [EGL|GLESv1_CM|GLESv2]}_*.so, если и этих библиотек нет, то загружается стандартная библиотека libGLES_android.so, которая реализует программную эмуляцию OpenGL ES 1.0 и EGL. Далее , используя функцию void *dlsym(void *restrict *handle*, const char *restrict *name*), в динамически подгруженных библиотеках находятся и заполняются указатели на соответствующие функции в структуре egl_connection_t. При следующем обращении к EGL API или OpenGL ES API, в случае успешной предыдущей загрузки функций, стандартная библиотека libEGL.so сначала преобрабатывает аргументы, вызываемых функций(например, проверяет их на корректность), а потом вызывает

соответствующую ранее подгруженную функцию из структуры egl_connection_t. Описанный выше механизм можно показать на наглядной схеме:



Первоначально планировалось полностью подменить существующую программную эмуляцию OpenGL ES на собственную реализацию библиотеки. В итоге видно, что для того, чтобы библиотека , реализующая виртуализацию OpenGL ES и EGL API правильно находилась, загружалась и внедрялась в систему, необходимо:

1. назвать библиотеку нужным образом

   (lib [EGL|GLESv1_CM|GLESv2]}_*.so)
2. расположить её в папке /vendor/lib/egl/

Так было сделано. Однако, как было описано выше, ANGLE поддерживает OpenGL ES v2 и EGL, а некоторые компоненты Android ОС (например, анимация загрузки операционной системы) используют OpenGL ES v 1.0. В результате, подход полной замены библиотеки, программно эмулирующей OpenGL ES v 1.0, был немного изменен. Об этом подробнее будет описано в следующем разделе.

3.3.2 Встраивание своей OpenGL ES и EGL библиотеки

Как писалось в предыдущем разделе, подход полной замены библиотеки, реализующей функционал OpenGL ES v 1.0 , на библиотеку, реализующей OpenGL ES v2.0 - невозможен. Тогда был исследован и выполнен другой подход.

Как писалось ранее, чтобы начать пользоваться OpenGL ES, необходимо настроить, используя EGL API, среду выполнения. В частности, необходимо создать контекст выполнения. Согласно спецификации EGL, если явно не указывать версию создаваемого контекста(для OpenGL ES v 1.x или OpenGL ES v 2.x), то по умолчанию контекст создается для версии 1.x . Иначе говоря, чтобы использовать OpenGL ES v 2.x, необходимо явно передать в функцию

```
EGLBoolean
eglChooseConfig(        EGLDisplay display,
                        EGLint const * attrib_list,
                        EGLConfig * configs,
                        EGLint config_size,
                        EGLint * num_config);
```

атрибут EGL_CONTEXT_CLIENT_VERSION со значением равным 2, что соответствует созданию контекста выполнения для OpenGL ES v 2.x. К тому же ANGLE, как писалось ранее, позволяет транслировать только OpenGL ES v 2.x в DirectX. Однако до исполнения этой функции невозможно понять какая версия OpenGL ES требуется приложению. Поэтому процесс подмены библиотеки был сделан следующим образом: любое приложение, которое использует OpenGL ES , изначально загружает библиотеку реализующую программную реализацию OpenGL ES, потом , когда выполняется функция eglChooseConfig(), в системной библиотеке libEGL идет проверка, на наличие атрибута EGL_CONTEXT_CLIENT_VERSION со значением равным двум. Если требуется версия контектса 2.x , тогда нужно выгрузить библиотеку, реализующую

программную эмуляцию, а потом загрузить свою библиотеку. Для этого сначала вызывается

```
EGLBoolean
eglTerminate(          EGLDisplay
                       display);
```

для текущего EGLDisplay, а потом вызвается

```
EGLBoolean
eglInitialize(         EGLDisplay display,
                       EGLint * major,
                       EGLint * minor);
```

В котором гостевая ОС непосредственно подключается к хостовому сервису. Этот процесс будет описан подробнее в следующих разделах[3].

3.3.3 Архитектура встраиваемых OpenGL ES и EGL библиотек

Ранее был описан интерфейс канала передачи данных, способ трансляции OpenGL ES в DirectX и способ встраивания своей версии OpenGL ES библиотеки. В этом разделе будет описана архитектура новой библиотеки OpenGL ES и EGL.

Суть виртуализации OpenGL ES и EGL заключается в том, что, когда программа обращается к этому API, вызовы параллельно исполняются на гостевой и на хостовой операционной системе. В большинстве случаев суть гостевой библиотеки передать аргументы вызываемой функии на хостовую ОС, исполнить там эту функцию, дождаться результата и передать его обратно. Эта часть легко реализуется используя VirtualBox HGCM, описанный ранее. В данном случае, в силу того, что хостовой операционной системой является Windows 8.1 , поэтому приходится транслировать вызовы OpenGL ES в DirectX, то основная роль хостового HGCM сервиса заключается в правильной обработке пришедших от гостя аргументов функций и вызова соответствующей функции ANGLE.

Однако, даже в этой части есть определенные трудности. Эти трудности связаны с тем, что , как писалось ранее, OpenGL ES - своеобразный конечный автомат: его контекст создается при помощи EGL, а состояния меняются путем вызова функций OpenGL ES API. Как только был установлен какой-либо контекст в качестве текущего, пока явно он не будет сменен другим, все функции OpenGL ES API будут производиться над этим контекстом. Можно представить ситуацию, когда параллельно запускаются два приложения, которые используют OpenGL ES. Сначала первое настраивает OpenGL ES, потом делает свой контекст текущем. В другом процессе аналогично поступает другое приложение. Если это все работает в рамках одной системы, то проблем не возникает: поскольку приложения запущены в разных процессах, то все вызовы OpenGL ES API происходят над локальными контекстами. Однако, если же эти приложения запускаются на гостевой системе, а обращения к OpenGL ES API перенаправляются на хостовую ОС, то вместе с этим теряется информация о том, что изначально приложения были запущены в рамках двух разных гостевых процессов, в результате, когда второе приложение сделает свой контекст

текущим, то вызовы первого приложения и второго будут менять состояние второго контекста, что , конечно, недопустимо.

Чтобы избежать эту проблему на хостовую операционную систему помимо передачи аргументов вызываемой функции передается идентификатор текущего потока рендеренга. Далее когда хостовый HGCM сервис получает этот идентификатор, от делает соответствующий контекст текущим и пришедшая функция меняет состояние нужного контекста.

Поскольку смысл виртуализации OpenGL ES заключается в грамотном делегировании всей работы по рендеренгу изображений хостовой видео-карте, то гостевая библиотека должна также заниматься контролированием этого процесса. Об этом подробнее будет написано в следующих разделах.

3.3.3.1 Особенности виртуализации EGL

Как писалось ранее , основными функциями EGL является создание контекста для OpenGL ES, связь его с текущей операционной системой, создание поверхностей для рисования, а так же показ на экране отрендеренного изображения. Т.к. EGL полностью заменяется, то и эти функции нужно реализовывать самостоятельно.

Однако из-за того, что рендеринг будет происходить на хостовой операционной системе, значит, и контекст, и инициализация, и создание поверхностей для рисования должны происходить на хостовой операционной системе. Собственно так и происходит: во время инициализации гостевой EGL, инициализируется хостовая ОС: узнаются доступные конфигурации EGL и экраны для отображения, происходит явное указание, что будет использоваться OpenGL ES, а далее заполняются соответствующие структуры этой информацией. После этого последующие гостевые вызовы, цель которых

получить существующие конфигурации или выбрать дисплей, будут идти не на прямую в ядро OpenGL , а будут обращаться к ранее заполненным структурам.

Непосредственная связь OpenGL ES API с оконной системой операционной системы происходит во время вызова

EGLBoolean eglSwapBuffers(EGLDisplay display, EGLSurface surface);

Сначала запрашивается у оконной системы свободный буфер, потом из хостовой системы берется уже отрендеренное изображение и оно копируется в запрошенный ранее буфер. Потом уже заполненный буфер передается оконной системе ОС и он отображается на дисплее устройства.

3.3.3.2 Особенности виртуализации OpenGL ES

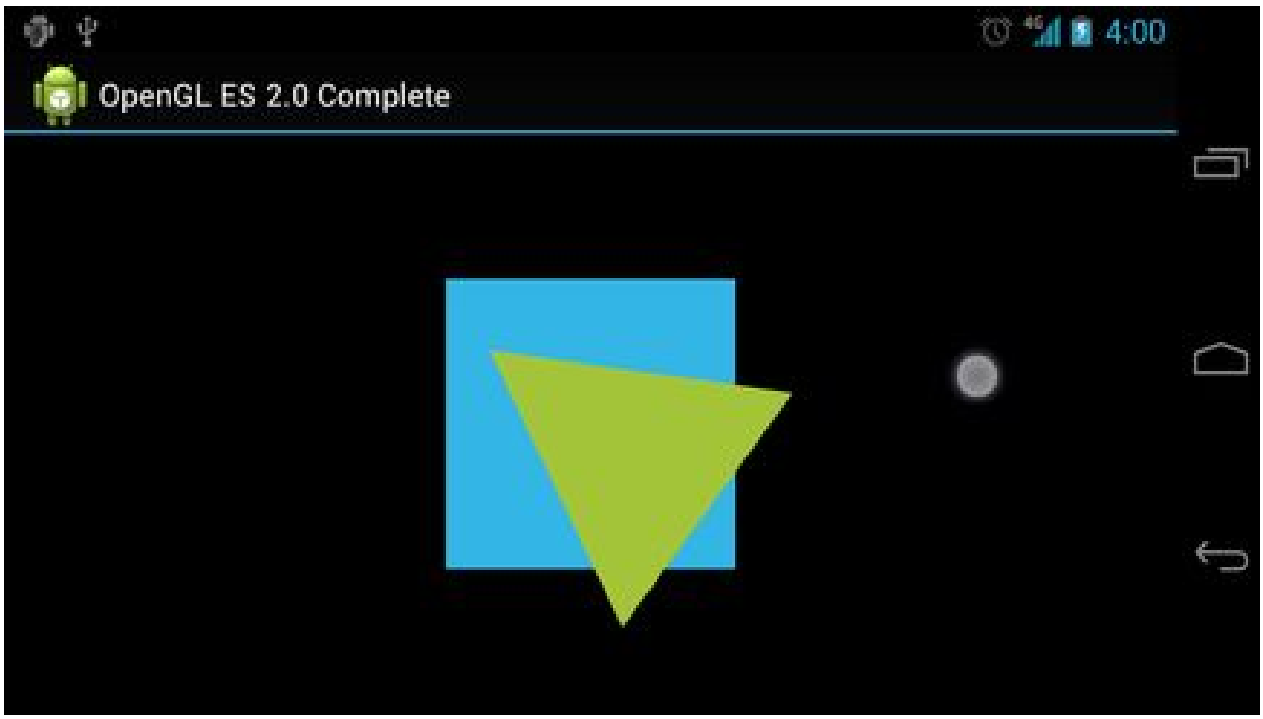
Для виртуализации большинства функций OpenGL ES API достаточно передать их на хостовую систему, а потом вызвать соответствующую функцию. Однако, есть функции, для которых этого недостаточно: в некоторые функции передается указатель на буфер, однако конкретный размер используемой в нем памяти передается в другом вызове. Для того, чтобы буфер корректно передался на хостовую ОС необходимо явно устанавливать размер передаваемого буфера. Поэтому делается следующее: метод , который явно устанавливает указатель на память не передается на хостовую ОС напрямую, вместо этого запоминаются все его аргументы на гостевой ОС, а потом , в вызове , в который передается точный размер буфера, они передаются на хостовую ОС вместе с новыми аргументами и там выполняются эти функции в паре.

Глава 4

Результаты

В рамках дипломной работы был разработан механизм виртуализации OpenGL ES. Были написаны библиотеки, которые реализуют часть функционала OpenGL ES и EGL API. Изучен и внедрен механизм передачи данных между гостевой и хостовой ОС: HGCM. На хостовой операционной системе написан HGCM сервис, который используя ANGLE транслирует OpenGL ES вызовы в DirectX.

Чтобы проверить работоспособность этой системы было написано простое приложение под Android, которое использует OpenGL ES API. На дисплей оно выводит треугольник, которые вращается по часовой стрелке:



Простые замеры fps показали около 60 кадров в секунду.

Заключение

В данной выпускной квалификационной работе был представлен подход к виртуализации OpenGL ES и EGL в Android ОС. Были исследованы и подробно изучены различные каналы передачи данных между гостевой и хостовой ОС: изучен и реализован протокол VirualBox Host-Guest Communication Manager.

Был найден и изучен способ трансляции вызовов OpenGL ES API в DirectX: Almost Native Graphics Layer Engine (ANGLE). Встроен он в хостовый HGCM - сервис.

Были исследованы различные подходы к подмену OpenGL ES библиотеки на Android ОС. Один из них был реализован.

Были написаны библиотеки, которые исспользуя канал передачи данных и OpenGL ES транслятор, перенаправляет вызовы из гостя в хостовую ОС, там их исполняет, а результат обратно возвращает в гостевую ОС.

В ходе выполнения данной работы была достигнута главная цель выпускной квалификационной работы: удалось написать инфраструктуру для виртуализации графической подсистемы Android ОС на уровне OpenGL ES, однако было реализовано не все его API. Реализация оставшегося API и его оптимизация планируется в будущем.

Список литературы

[1] VirtualBox.org https://www.virtualbox.org/

[2] Android-x86 Open Source Project Announcement - http://www.android-x86.org/

[3] OpenGL ES - The Standard for Embedded Accelerated 3D Graphics

- https://www.khronos.org/opengles/

[4] Graphics architecture

https://source.android.com/devices/graphics/architecture.html

[5] ANGLE - Almost Native Graphics Layer Engine

- https://github.com/google/angle

[6] Android Open Source Project https://source.android.com/

[7] VirtualBox Programming Guide and Reference

http://download.virtualbox.org/virtualbox/4.1.20/SDKRef.pdf

[8] The ANGLE Project: Implementing OpenGL ES 2.0 on Direct3D // Pages:

543–570. Citation: Daniel Koch and Nicolas Capens.

[9] OpenGL® ES 2.0 Programming Guide // ISBN-13: 078-5342502794

ISBN-10: 0321502795