DOI: 10.17725/rensit.2024.16.407

Обеспечение функционирования графических GLX приложений на промышленном оборудовании с использованием API EGL

Татарчук И.А., Мамросенко К.А., Гиацинтов А.М.

НИИ системных исследований РАН, Центр визуализации и спутниковых информационных технологий, https://niisi.ru/

Москва 117218, Российская Федерация

E-mail: tatarchuk_ia@niisi.ras.ru, mamrosenko_k@niisi.ras.ru, algts@niisi.ras.ru

Поступила 12.03.2024, рецензирована 19.03.2024, принята 23.03.2024

Представлена действительным членом РАЕН А.С. Дмитриевым

Аннотация: **В работе рассматриваются вопросы функционирования программного обеспечения для решения инженерных задач в условиях ограниченной поддержки драйверами графического адаптера отдельных подсистем графического стека OS Linux. Показано, что инженерное и управляющее промышленным оборудованием ПО, подсистема визуализации которого использует API GLX будет требовать доработки для переноса на встраиваемые системы, использующие графические ускорители с драйверами поддерживающими только API EGL. Целью исследования является разработка подходов для обеспечения функционирования программного обеспечения, применяющегося на промышленном оборудовании и использующего API GLX, через API EGL в графической подсистеме ОС Linux. Рассмотрен метод трансляции вызовов GLX API в EGL API, определена его применимость. Разработан новый алгоритм организации взаимодействия между пользовательскими программами и графической подсистемой ОС Linux, позволяющим запускать приложения, использующие GLX API в условиях отсутствия поддержки DRI драйвером графического ускорителя. Проведено тестирование корректности работы алгоритма и проведено сравнение с результатами тестирования открытых драйверов с поддержкой DRI2. Полученные результаты позволят сократить затраты ресурсов на поддержку отдельных подсистем графических драйверов и осуществить "бесшовный" переход на использование EGL API во встраиваемых системах.**

Ключевые слова: **GLX, EGL, Xorg, Linux**

УДК 004.454

Благодарности: Работа выполнена в рамках НИР ФГУ ФНЦ НИИСИ РАН по теме № FNEF-2024-0003 "Методы разработки аппаратно-программных платформ на основе защищенных и устойчивых к сбоям систем на кристалле и сопроцессоров искусственного интеллекта и обработки сигналов".

Для цитирования: Татарчук И.А., Мамросенко К.А., Гиацинтов А.М. Обеспечение функционирования графических GLX приложений на промышленном оборудовании с использованием API EGL. *РЭНСИТ: Радиоэлектроника. Наносистемы. Информационные технологии*, 2024, 16(3):407-418. DOI: 10.17725/rensit.2024.16.407.

Graphical GLX-applications functioning ensuring on industrial equipment using the EGL API

Ivan A. Tatarchuk, Kirill A. Mamrosenko, Alexander M. Giatsintov

Scientific Research Institute of System Analysis of the RAS, Center of Visualization and Satellite Information Technologies, https://niisi.ru/

Moscow 117218, Russian Federation

E-mail: tatarchuk_ia@niisi.ras.ru, mamrosenko_k@niisi.ras.ru, algts@niisi.ras.ru

Received March 12, 2023, peer-reviewed March 19, 2023, accepted March 23, 2023.

Abstract: **This paper discusses the operation of engineering software on systems with graphics adapter drivers that provide limited to no support for certain graphics subsystems of Linux OS. The paper demonstrates that software with GLX API-based visualization subsystem needs modification to be ported to the embedded systems using graphics accelerators with drivers that support only the EGL API. The purpose of the study is to develop approaches for enabling the operation of software on industrial equipment and utilizing the GLX API via the EGL API of the Linux-based graphics subsystem. The paper discussed the method for translating GLX API calls to the EGL API and its applicability. A new algorithm has been developed for organizing interaction between user applications and the Linux OS graphics subsystem, which allows running applications with the GLX API in the absence of DRI support by the graphics accelerator driver. The correctness of the algorithm's operation has been tested and compared with the results of the tests of open drivers with DRI2 support. The findings will make it possible to reduce the resources needed to support individual graphics driver subsystems and make a smooth transition to using the EGL API in the embedded systems.**

Keywords: **GLX, EGL, Xorg, Linux**

UDC 004.454

Acknowledgments: The work was carried out as part of national assignment for SRISA RAS (on the topic No. FNEF-2024-0003.

For citation: Ivan A. Tatarchuk, Kirill A. Mamrosenko, Alexander M. Giatsintov. Graphical GLX-applications functioning ensuring on industrial equipment using the EGL API. *RENSIT: Radioelectronics. Nanosystems. Information Technologies*, 2024, 16(3):407-418e. DOI: 10.17725/j.rensit.2024.16.407.

Содержание

1. ВВЕДЕНИЕ

В настоящее время все больше микропроцессоров для встраиваемых систем оснащаются графическим ускорителем. Благодаря росту производительности, усложняются и задачи, решаемые на данных системах. Такие микропроцессоры применяются при разработке приборных панелей, планшетных компьютеров, элементов систем интерфейса оператора промышленного оборудования и т.д. Для визуализации, к примеру, деталей, чертежей, схем производственных процессов на устройствах отображения промышленного оборудования зачастую необходимо использовать динамическое трёхмерное окружение, с частотой обновления не менее 25 кадров в секунду. В этом случае графический ускоритель должен иметь функции аппаратного ускорения 3D графики. Чаще всего последнее подразумевает наличие поддержки API OpenGL и OpenGL ES.

Взаимодействие программ, использующих ускорение вывода трехмерной графики и операционной системы происходит не напрямую, а через несколько слоев абстракции графической подсистемы, каждый из которых имеет свой программный интерфейс. По историческим причинам, в ОС Linux для этого используется GLX (OpenGL's X extension) – API,

позволяющий использовать команды OpenGL в приложениях для X Window System.

Графический ускоритель в микропроцессоре может представлять собой проприетарный готовый СФ блок, драйвер которого поставляется разработчиком. В ряде случаев разработчики ориентируются на то, что их графический ускоритель будет использоваться в устройствах с мобильной ОС Android, поэтому поставляемый драйвер имеет реализацию интерфейса EGL (который есть и в Linux). EGL является основным интерфейсом доступа к графической подсистеме ОС Android, поскольку эта ОС не использует X Window System и GLX соответственно.

Поскольку ряд программных продуктов для визуализации, применяемых в промышленном оборудовании, CAD-систем (напр. kiCAD) используют GLX, то появляется необходимость доработки ПО под API EGL, для запуска с драйверами без поддержки GLX. Последнее потребует определенных трудозатрат и в ряде случаев повторной сертификации.

Исходя из вышеизложенного следует, что представляет интерес разработка таких методов организации взаимодействия между пользовательскими программами и графической подсистемой ОС Linux, которые позволят запускать приложения, использующие GLX, через подсистему EGL. В последнее время, в публикациях появились сведения о новых методах организации взаимодействия пользовательских программ и графической подсистемы Linux, позволяющих запускать GLX приложения посредством API EGL. При этом описание методов не детализированное и требуется практическая проработка по внедрению их в программное обеспечение, реализующее графическую подсистему Linux.

Научная новизна настоящей работы заключается в разработанном алгоритме организации взаимодействия между пользовательскими программами и графической подсистемой ОС Linux, позволяющем запускать приложения, использующие GLX API в условиях отсутствия поддержки DRI драйвером графического ускорителя.

Настоящее исследование – попытка ответить на вопрос, каким образом обеспечить работу приложений, использующих API GLX, через API EGL в графической подсистеме ОС Linux.

В разделе 2 приводится обзор работ, посвященных принципам организации и работе графической подсистемы ОС Linux. В разделе 3 приводится описание существующей архитектуры графической подсистемы и особенности запуска приложений в ней. В разделе 4 описывается разработанный алгоритм взаимодействия. Результаты практической проверки работы алгоритма и сравнительный анализ функциональных возможностей по запуску графических приложений приведены в разделе 5. Выводы и предложения по дальнейшему развитию приведены в Заключении.

2. АНАЛИЗ ПРЕДШЕСТВУЮЩИХ РАБОТ

По вопросам модернизации, защищенности, и оптимизации таких составных частей графической подсистемы ОС Linux как DRI и GLX исследования проводятся на протяжении последних 20 лет. В работе [1] авторы исследуют возможность по виртуализации ресурсов аппаратного видеоускорителя в веб-браузерах, при этом реализуя API GLX и API EGL для клиентских приложений, предпочитая первый последнему. Проводятся исследования по вопросам переноса с API GLX на API EGL приложений визуализации молекулярных соединений на суперкомпьютерах [2], для сокращения накладных расходов на работу X сервера. Вопрос реализации API GLX и способов оптимизации взаимодействия его с X сервером рассмотрены в [3], где авторы задаются вопросом сохранения состояний GLX контекста для обеспечения отказоустойчивости пользовательских приложений. Использование API EGL для написания приложений для встраиваемых систем на основе Linux освещено в [4] и [5]. Особенности портирования приложений, написанных на EGL API под Android на Linux для встраиваемых систем, возникающие из-за отличий в графическом стеке, рассматриваются в [6]. В работе [7] авторы показывают особенности реализации EGL в мобильных встраиваемых системах.

3. ГРАФИЧЕСКАЯ ПОДСИСТЕМА ОС LINUX И ВЗАИМОДЕЙСТВИЕ ПОЛЬЗОВАТЕЛЬСКИХ ПРИЛОЖЕНИЙ С НЕЙ

3.1. Список терминов

Для удобства восприятия текста настоящей работы, мы предлагаем следующие понятия:

- GLX – спецификация, разработанная SGI, и описывающая расширение OpenGL для X Window System.
- X server – реализация X Window System от разработчиков проекта X.org
- GLX API – программный интерфейс приложений, позволяющий использовать GLX. Через GLX API приложения получают доступ к возможностям аппаратного ускорения трехмерной графики.
- EGL – Khronos Native Platform Graphics Interface (платформонезависимый графический интерфейс). Аббревиатура EGL не расшифровывается. EGL является программным интерфейсом для доступа к возможностям аппаратного ускорения трехмерной графики из операционной системы.

3.2. Архитектура подсистемы DRI и взаимодействие графических приложений с ней

Подсистема DRI – не отдельная, в смысле группировки её исходного кода, подсистема Linux, как DRM (direct rendering manager), а совокупность интерфейсов между X server, драйвером графического ускорителя и самой DRM. Изначально подсистема DRI была разработана как часть проекта Xorg по созданию единого интерфейса доступа к ресурсам видеоускорителя, однако, впоследствии, DRI стала основой графической подсистемы Linux. Существует несколько версий подсистемы DRI – 1, 2 и 3. Первая версия считается устаревшей, поскольку в ней доступ к аппаратным ресурсам видеоускорителя предоставлялся одному приложению в исключительном порядке. Третья версия протокола значительно упростила многие операции, однако, подсистема GLX в X server по прежнему использует протокол второй версии. Поэтому в настоящей работе мы будем исследовать именно DRI2. В качестве объекта для изучения непосредственно реализации графического драйвера с DRI2 мы возьмем драйверы из проекта MESA.

Как видно из **Рис. 1** DRI включает в себя подсистему DRM [8], подсистему DRI графического драйвера и отдельную сущность под названием GLXProvider. GLX API реализуется двумя сущностями – подсистемой GLX в X server и подсистемой GLX драйвера. Подсистема GLX драйвера и подсистема GLX в X server не входят в DRI, но предоставляют интерфейс к ней. Каждая из представленных подсистем и сущностей, за исключением GLXProvider, представлена обособленной единицей трансляции, в виде разделяемой библиотеки. Взаимодействие между подсистемами GLX X server и подсистемой GLX драйвера происходит посредством протокола X11 и прямым вызовом функций из разделяемых библиотек драйвера [9]. Описание и особенности работы каждого из элементов архитектуры мы рассмотрим более подробно далее.

Для понимания принципов функционирования DRI2 следует вначале рассмотреть работу типового графического

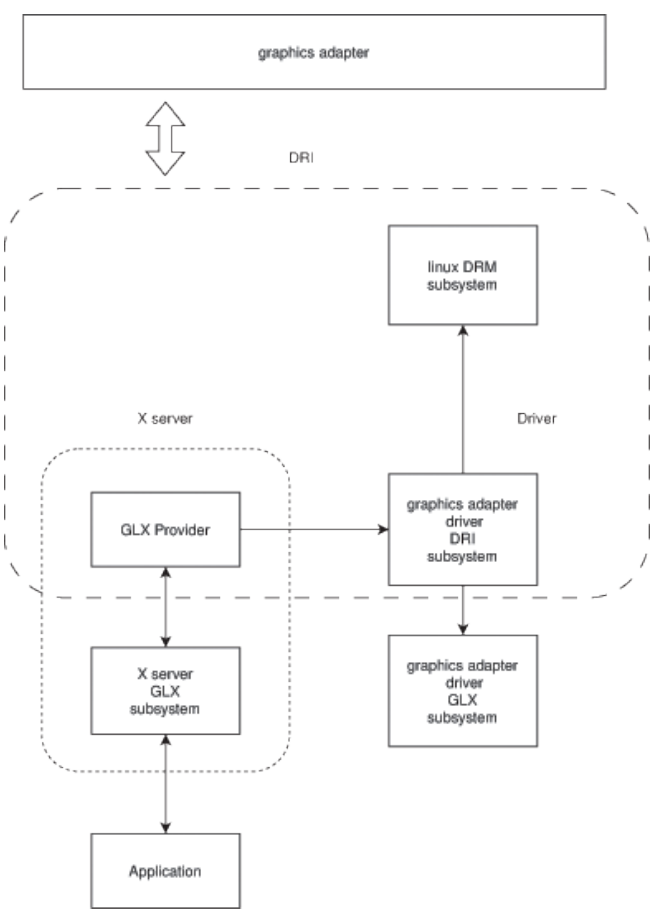

Рис. 1. *Архитектура подсистемы DRI и её компоненты.*

приложения, использующего GLX API. Приложение использует GLX API следующим образом:

1. Подключается к дисплею (display) X server (функция XOpenDisplay из Xlib).
2. Запрашивает список поддерживаемых графическим драйвером расширений GLX (функция glXQueryExtensionsString).
3. Запрашиваются настройки отображения (функция XVisualInfo).
4. Создается буфер для вывода изображения (в спецификации такой объект называется GLXdrawable).
5. Создается GLX контекст, с полученными на этапе 3 настройками отображения.
6. Контекст привязывается к буферу для вывода изображения.
7. Выполняется формирование и вывод изображения в буфер.
8. Контекст "разрушается" в конце работы программы.

Примечание: Расширения GLX – это дополнения к спецификации GLX, которые реализуют различный функционал драйверов по формированию изображений. Возможность добавлять и использовать такие расширения в драйверах графических ускорителей была введена в спецификации GLX версии 1.1. Производители видеоускорителей изначально закладывали свои собственные (проприетарные) расширения в драйверы. Некоторые из них стали фактически стандартными и в настоящее время их поддержка является обязательной.

Для создания контекста используется функция из GLX API - glXCreateContext. По соглашению, предложенному разработчиками X.org, далее внутри glXCreateContext происходит вызов функции __glXInitialize. Вызов __glXInitialize позволяет драйверу выполнить следующие действия:

- Запросить список расширений GLX подсистемы X сервера, и запросить версию DRI от GLX подсистемы X сервера;
- загрузить DRI части драйвера графического ускорителя;
- инициализировать структуру, описывающую дисплей, и передать ее приложению.

Имя разделяемой библиотеки, описывающей DRI подсистему драйвера обычно состоит из идентификатора типа "vendor_name" + "_dri.so". По соглашению разработчиков, подсистема DRI драйвера обязана иметь функции с именами _driCreateScreen и __driCreateNewScreen. При загрузке библиотеки в структуре, описывающей дисплей, передаются указатели на функции создания и удаления буферов отрисовки (glxDrawables) и т.д., которые будут вызываться при работе приложения с конкретным экраном в режиме аппаратного ускорения.

Для понимания реализации подсистем в GLX драйвере и X server следует рассмотреть работу и использование самого GLX API. К настоящему моменту времени в Linux используется так называемый общий (shared) графический стек [10]. Общий стек – это такой стек, в котором для каждого экрана (screen) X server привязывается отдельный графический драйвер (в случае если их несколько). Такая возможность обеспечивается библиотекой libglvnd. Название libglvnd представляет собой акроним - "GL vendor neutral dispatch library", что можно перевести как "библиотека трансляции функций API OpenGL, независящая от драйвера". Библиотека libglvnd предоставляет пользовательским программам возможность использовать единый бинарный интерфейс функций OpenGL, GLX и OpenGLES, путем составления таблицы трансляций вызовов функций из своих разделяемых библиотек в аналогичные функции, который предоставляют драйверы производителей видеоускорителей.

Схема взаимодействия приложения, libglvnd и разделяемых библиотек драйверов, показана на **Рис. 2**.

Когда приложение обращается к GLX API, оно обращается к функциям GLX API (описанных в glx.h) из разделяемой системной библиотеки libGLX.so, которая является частью libglvnd. После вызова функции из GLX API,

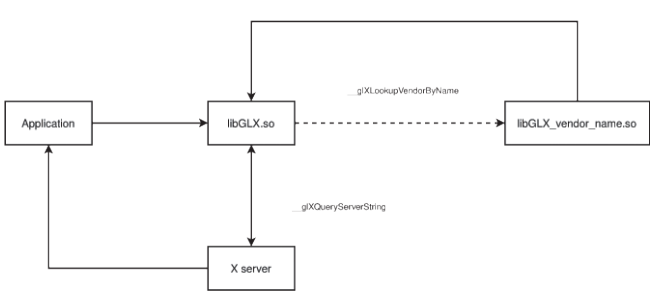

Рис. 2. *Функциональная схема взаимодействия приложения с libglvnd.*

внутри libGLX.so происходят следующие действия:

- При загрузке библиотеки, посредством расширения компилятора GCC "__attribute__ ((constructor))", проверяется наличие переменной окружения "__GLX_VENDOR_LIBRARY_NAME", которая определяет имя разделяемой библиотеки подсистемы GLX драйвера.
- Если такая переменная окружения не существует, то при вызове функции из GLX API происходит процедура определения значения переменной glx-vendor. Эта переменная хранит имя драйвера графического ускорителя, которое чаще всего совпадает с именем компании производителя. Процедура определения значения glx-vendor заключается в общем случае в следующем: получив от X server данные о дисплее и экране, libglvnd извлекает значение glx-vendor через вызов функции __glXQueryServerString с параметром GLX_VENDOR_NAMES_EXT.
- Определив значение glx-vendor, библиотека libglvnd осуществляет загрузку разделяемой библиотеки конкретного драйвера, которая определяет реализация подсистемы GLX драйвера. Через функцию __glXLookupVendorByName загружается конкретная разделяемая библиотека, после чего libglvnd и формирует таблицу с указателями на функции с реализацией GLX API в подсистеме GLX драйвера.

3.3. Взаимодействие DRI и X server и драйвера графического ускорителя

Как видно из Рис. 2 X server определяет какой графический драйвер привязан к конкретному экрану. Можно сделать вывод, что X server содержит в себе GLX сервер, а приложение, использующее GLX API, является клиентом. В этом случае для полного понимания работы GLX в DRI2 будет полезным описать, как X server привязывает конкретный графический драйвер к определенному экрану [11].

Известно, что архитектура X server является расширяемой. Расширяемость организуется за счет подгрузки разделяемых библиотек, реализующих ту или иную функциональность. Такие библиотеки называются модулями X server. Модуль GLX, представляющий собой подсистему GLX сервера, реализован в разделяемой библиотеке libglx.so. Программный интерфейс библиотеки libglx.so представлен структурой glxModuleData типа "XF86ModuleData". Структура содержит указатель на функцию инициализации модуля – glxsetup. Функция glxsetup выполняет поиск экземпляра структуры GLXprovider с именем __glXDRI2Provider. Структура GLX_Provider – это абстрактный класс данных, который предоставляет X серверу интерфейс для установления взаимодействия с подсистемой GLX конкретного драйвера. __glXDRI2Provider хранит указатель на функцию подключения экрана (в настоящей реализации она имеет имя glxScreenProbe). При обнаружении экземпляра структуры с обозначенным именем она добавляется в связный список через функцию GlxPushProvider, в котором хранятся ссылки на иные возможные экземпляры GLXprovider.

После нахождения экземпляра GLXprovider, вызывается функция xorgGlxCreateVendor, которая добавляет указатель на функцию xorgGlxServerInit в связный список функций, которые будут вызываться при загрузке подсистемы GLX X server. Можно считать что после этого происходит регистрация GLXprovider в качестве единственного интерфейса к подсистеме GLX. На этом этапе X server выводит сообщение о том, что расширение подключено, но еще не проинициализировано: "Extension "GLX" is enabled". *Примечание*: под подключением расширения, следует понимать, что X server осуществил успешную загрузку модуля, что означает, что функция инициализации модуля найдена, но её вызов пока отложен.

После подключения и инициализации всех оставшихся расширений, X server начинает инициализацию модуля GLX, и, соответственно, инициализацию подсистемы GLX посредством вызова функции xorgGlxServerInit. В этой функции происходит создание и привязка экрана к конкретному графическому драйверу. Перед этим происходит проверка на то, что инициализация хотя бы одного GLXprovider из списка уже произведена. Создание нового экрана происходит через вызов функции glxScreenProbe из структуры __glXDRI2Provider, и осуществляется следующим образом:

- Выделяется память под новый экземпляр абстрактного класса, описывающего экран, который создается через DRI;
- Для экрана, осуществляется привязка функций по созданию экрана, буферов и их удалению.
- Проверяется наличие возможности использования DRI2 для нового экрана, и имя драйвера графического ускорителя, поддерживающего работу по DRI2.
- Загружается расширяемая библиотека с подсистемой glx драйвера.
- Вызывается функция создания нового экрана из GLX подсистемы драйвера.

Если экран успешно создан, выводится сообщение "AIGLX: Loaded and initialized <driverName>". Разработчики X server ввели термин AIGLX (accelerated indirect) GLX, которым описывается способ организации взаимодействия GLX подсистемы X server и подсистемы DRI. К настоящему моменту времени AIGLX является единственным способом их взаимодействия. После создания экрана, X server выводит сообщение "GLX: Initialized DRI2 GL provider for screen 0", которое говорит, что для экрана с определенным номером осуществляется графическое ускорение посредством GLX_Provider с именем "DRI2 GL". Аналогично для программного рендеринга GLX_Provider имеет название IGLX (indirect GLX).

Так описывается взаимодействие приложения, использующего GLX API, подсистемы GLX X server и подсистемы GLX графического драйвера. Поскольку в настоящем описании не затронуто взаимодействия с DRI с DRM, то, для полноты понимания графического стека Linux, следует описать и этот аспект.

При загрузке модуля GLX сам X server уже имеет сведения об имени графического ускорителя, чей драйвер поддерживает работу с DRI. Эти сведения X server получает при загрузке модуля modesetting. При загрузке модуля modesetting происходит инициализация компонентов DRI2 в X server. В процессе загрузки модуля modesetting вызывается функция, чей адрес расположен в переменной pScrn->ScreenInit. В этой функции происходит инициализация подсистемы аппаратного ускорения glamor (при ее задействовании). Glamor дает X server возможность использовать аппаратное ускорение вывода трехмерной графики при отображении элементов двумерного интерфейса. Glamor представляет собой отдельный модуль X server, и позволяет самому X server создавать буферы для вывода изображений окон и т.д. Glamor может работать как через API OpenGL, так и через API OpenGL ES[10].

После этого начинается инициализация подсистемы DRI2, которая проходит следующим образом:

- Если драйвер поддерживает EGL (что необходимо для запуска glamor), то имя драйвера запрашивается посредством расширения EGL_MESA_query_driver.
- Если такого расширения нет, то X server определяет имя устройства видеоускорителя сам, посылая запрос в DRM (поиск идет по имени в дереве устройств /dev, которое прописывается в настроечный файл xorg.conf).
- Проверяется факт авторизации (проверка наличия уникального числа – ключа для доступа к аппаратным ресурсам устройства) драйвера в DRM согласно DRI2.
- Определяются допустимые параметры экрана.
- По завершении инициализации DRI2 выводится сообщение "[DRI2] Setup complete".

Исходя и вышеописанного можно сделать следующие выводы:

- В настоящее время X server по умолчанию использует EGL для запуска подсистемы аппаратного ускорения glamor.
- Подсистема DRI2 запускается уже после инициализации glamor и представляет собой слой абстракции для загрузки и привязки драйвера графического ускорителя к экрану.
- Пользовательские графические приложения, использующие GLX API, получают доступ к ресурсам графического ускорителя посредством X server.
- Драйверу графического ускорителя для обеспечения GLX API пользовательских приложений, необходимо иметь две расширяемые библиотеки с реализацией GLX и DRI подсистем.

Исходя из приведенного выше, можно формализовать алгоритм взаимодействия

между пользовательскими приложениями и графической подсистемой ОС Linux. Алгоритм который используется в настоящий момент времени выглядит так:

Шаг 1. Если во время запуска X server, при загрузке модуля GLX X server, представляющего собой разделяемую библиотеку, в нем обнаружена структура __glXDRI2Provider, то поместить указатель на неё в связный список, содержащий указатели на GLXprovider, иначе перейти на шаг 5.

Шаг 2. Произвести поиск драйвера графического ускорителя, поддерживающего DRI2, и загрузить разделяемую библиотеку, содержащую GLX подсистему драйвера.

Шаг 3. Создать экран (screen) с допустимыми параметрами посредством функции создания экрана из подсистемы GLX драйвера.

Шаг 4. Вывести сообщение об успешной загрузке подсистемы GLX драйвера посредством AIGLX и перейти к шагу 6.

Шаг 5. Осуществить загрузку структуры __ glXDRISWRastProvider, которая предоставит возможность загрузить подсистему GLX драйвера, реализующего программный рендеринг.

Шаг 6. Запустить приложение, использующее GLX API

4. РАЗРАБОТКА АЛГОРИТМА ВЗАИМОДЕЙСТВИЯ МЕЖДУ ПОЛЬЗОВАТЕЛЬСКИМИ ПРИЛОЖЕНИЯМИ И ГРАФИЧЕСКОЙ ПОДСИСТЕМОЙ ОС LINUX

Повсеместное распространение ОС Android послужило причиной того, что при разработке графических драйверов для видеоускорителей, производители не обеспечивают поддержку GLX и DRI, ограничиваясь реализацией поддержки EGL. В этом случае использование большей части графических приложений под X server становится затруднительным. При этом, сам X server использует EGL для ускорения отрисовки оконного интерфейса через glamor.

Для решения проблемы запуска Linux приложений, написанных с использованием GLX, с драйверами, в которых не реализована подсистема DRI, американский исследователь Адам Джексон (Adam Jackson) предложил использовать метод трансляции вызовов GLX API в EGL API, который имеет название "GLX-EGL adapter". Такой метод Адам Джексон уже использовал в Xwayland, проекте по обеспечению совместимости X server и оконной системы Wayland.

Суть метода "GLX-EGL adapter" заключается в следующем: основная функция glxScreenProbe заменяется egl_screen_probe, в которой для создания экрана и буферов обращение идет не к подсистеме DRI драйвера, а к EGL. Таким образом, для работы GLX приложения необходима лишь GLX подсистема драйвера, а DRI подсистема не требуется – см. **Рис. 3.**

Создание нового экрана в функции egl_ screen_probe осуществляется следующим образом:

- Выделяется память под новый экземпляр абстрактного класса, описывающего экран, которые создается через glamor.
- Для экрана, осуществляется привязка функций по созданию экрана, буферов и их удалению. Здесь следует отметить одну из главных особенностей решения, предлагаемого Джексоном: в функции для создании буфера вывода и т.п. осуществляется привязка к EGL API вместо GLX API.

Таким образом, благодаря методу "GLX-EGL adapter" отпадает необходимость в реализации DRI подсистемы драйвера.

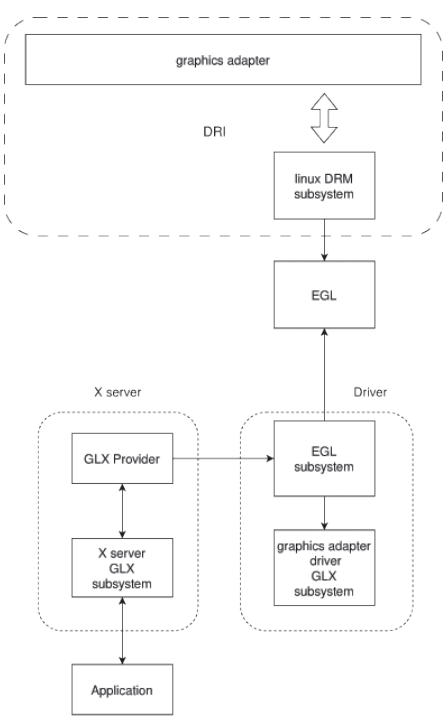

Рис. 3 *Архитектура графического стека при использовании "GLX-EGL adapter".*

Предложенный Джексоном метод описан как потенциально реализуемый, но не приводятся конкретные алгоритмы, способы их реализации в коде X server, результаты тестирования. Поэтому в настоящей работе приведен алгоритм организации взаимодействия между пользовательскими программами и графической подсистемой ОС Linux, позволяющий запускать приложения, использующие GLX API в условиях отсутствия поддержки DRI драйвером графического ускорителя.

В реализации метода "GLX-EGL adapter" создается еще один экземпляр структуры GLXprovider, под именем glamor_provider. Для того, что бы X server смог загрузить нужный нам GLXprovider, необходимо добавить glamor_provider в связанный список с помощью функции GlxPushProvider. Однако необходимо, чтобы glamor_provider оказался в начале списка, и далее происходил вызов xorgGlxCreateVendor. Очевидно, что регистрация glamor_provider должна происходить до регистрации __glXDRI2Provider. Поэтому следует определить конкретное место в коде X server, где стоит непосредственно производить регистрацию glamor_provider. Авторы считают, что регистрацию glamor_provider следует проводить при инициализации подсистемы glamor, до загрузки самого модуля GLX. Такой подход позволит снизить связность между единицами трансляции, в частности, модулями GLX и glamor и осуществлять инициализацию glamor_provider только тогда, когда используется сам glamor. Последнее актуально в том случае, если имеется необходимость отключать сам модуль glamor.

Исходя из вышеизложенных соображений по glamor_provider, следует внести изменения в уже существующий и изложенный выше алгоритм взаимодействия между пользовательскими приложениями и графической подсистемой ОС Linux. Таким образом, обновленный алгоритм, который авторы предлагают, выглядит так:

Шаг 1. Во время загрузки модуля glamor, поместить указатель на glamor_provider в связный список, содержащий указатели на GLXprovider.

Шаг 2. Создать экран (screen) с допустимыми параметрами посредством функции создания экрана из glamor.

Шаг 3. Вывести сообщение об успешной загрузке подсистемы GLX драйвера посредством glamor_provider.

Шаг 4. Запустить приложение, использующее GLX API.

Следует отметить, что предложенный алгоритм не нарушает функционирование предыдущего, в том смысле, что загрузка glamor_provider осуществляется до __glXDRI2Provider, и при отсутствии необходимости использовать glamor будет использоваться __glXDRI2Provider.

5. ПРОВЕРКА РАЗРАБОТАННОГО АЛГОРИТМА И ОБСУЖДЕНИЕ РЕЗУЛЬТАТОВ

Для проверки работоспособности метода, и соответственно, предложенного алгоритма, следует определить, что именно и как будет проверяться, а также как оценивать результаты. Процесс проверки работоспособности метода будет заключаться в проведении эксперимента по запуску GLX приложений. В такой тривиальной постановке эксперимента следует поднять вопрос – а успешный запуск какого приложения будет является достаточным для подтверждения результата? Авторы предлагают рассматривать проверку работоспособности предложенного алгоритма как тестирование ПО. Таким образом, следует составить тест-план для проведения тестирования X server с нашими изменениями. В качестве рекомендаций по написанию плана тестирования мы будем использовать стандарт IEEE 829-1998.

Согласно стандарту следует определить сам объект тестирования и функции которые будут протестированы. В нашем случает объектом тестирования является графическая подсистема ОС Linux. Мы будем тестировать новую функцию графической подсистемы – запуск приложений, использующих API GLX посредством glamor_provider. Последнее можно еще более конкретизировать – мы будем тестировать функцию взаимодействия подсистемы GLX X server и подсистемы GLX графического драйвера.

Далее стандарт предлагает выбрать подход к тестированию. Он заключается в выборе методики проведения тестирования и методики анализа результатов. Существует набор тестов

для GLX подсистемы драйвера – группа тестов категории GLX из набора тестов Piglit. Piglit – набор тестов для графических драйверов, написанных под ОС Linux, ранее входивший в состав проекта Mesa, а позже вынесенный из его кодовой базы. Каждый тест представляет собой приложение, использующее API GLX для проверки отельных положений спецификации.

Методика анализа результатов будет заключаться в сравнительном анализе результатов тестирования драйвера графического ускорителя (драйвер 1), который не имеет поддержки DRI2, и драйвера с открытым исходным кодом из проекта Mesa, результаты тестирования которого мы будет считать эталонными (драйвер 2).

В драйвере 1 реализована подсистема GLX API, схожая с реализацией в MESA, но не идентичная ей. В качестве драйвера 2 будет использоваться драйвер Nouveau из Mesa. Версия X server - 1.20.13.

Результаты тестирования приведены в **Таблице 1**.

Таблица 1

Результаты тестирования

Имя теста	Драйвер 1, результаты	Драйвер 2, результаты
glx-buffer-age	–	+
glx-close-display	+	+
glx-context-flush-control	–	–
glx-copy-sub-buffer	–	+
glx-create-context-both-es-strings	+	+
glx-create-context-core-profile	+	+
glx-create-context-current-no-framebuffer	–	+
glx-create-context-default-major-version	+	+
glx-create-context-default-minor-version	+	+
glx-create-context-ext-no-config-context	–	–
glx-create-context-indirect-es2-profile	–	+
glx-create-context-invalid-attribute	–	+
glx-create-context-invalid-es-version	–	–
glx-create-context-invalid-flag	–	–
glx-create-context-invalid-flag-forward-compatible	–	+
glx-create-context-invalid-gl-version	–	+
glx-create-context-invalid-profile	–	+
glx-create-context-invalid-render-type	–	+
glx-create-context-invalid-render-type-color-index	–	+
glx-create-context-invalid-reset-strategy	*	–
glx-create-context-no-error	–	–
glx-create-context-pre-GL32-profile	+	+
glx-create-context-require-robustness	*	+
glx-create-context-valid-attribute-empty	–	+
glx-create-context-valid-attribute-null	–	+
glx-create-context-valid-flag-forward-compatible	+	+
glx-destroycontext-1	+	+
glx-destroycontext-2	+	+
glx-destroycontext-3	+	+
glx-dont-care-mask	+	+
glx-egl-switch-context	+	+
glx-fbconfig-bad	+	+
glx-fbconfig-compliance	+	+
glx-fbconfig-sanity	+	+
glx-fbo-binding	+	+
glx-free-context	–	–
glx-get-context-id	–	–
glx-get-current-display-ext	–	–
glx-import-context-has-same-context-id	–	–
glx-import-context-multi-process	–	–
glx-import-context-single-process	–	–
glx-make-current	–	–
glx-make-current-multi-process	–	–
glx-make-current-single-process	–	–
glx-make-glxdrawable-current	+	+
glx-multi-context-front	–	+
glx-multi-context-ib-1	–	+
glx-multi-context-single-window	–	+
glx-multithread	–	+
glx-multithread-buffer	–	+
glx-multithread-buffer-refcount-bug	–	+
glx-multithread-clearbuffer	–	–
glx-multithread-makecurrent-1	–	–
glx-multithread-makecurrent-2	–	–
glx-multithread-makecurrent-3	–	–
glx-multithread-makecurrent-4	–	–
glx-multithread-shader-compile	+	–
glx-multithread-texture	+	–
glx-multi-window-single-context	+	–
glx-oml-sync-control-getmscrate	–	+
glx-oml-sync-control-swapbuffersmsc-divisor-zero	–	+
glx-oml-sync-control-swapbuffersmsc-return	–	+
glx-oml-sync-control-timing	–	+
glx-oml-sync-control-waitformsc	–	+
glx-pixmap13-life	–	+
glx-pixmap-crosscheck	–	+
glx-pixmap-life	+	+
glx-pixmap-multi	–	+
glx-query-context-info-ext	–	–
glx-query-drawable	+	+
glx-query-renderer-coverage	–	+
glx-shader-sharing	–	+

Таблица 1, продолжение

Результаты тестирования

Имя теста	Драйвер 1, результаты	Драйвер 2, результаты
glx-string-sanity	+	+
glx-swap-copy	–	+
glx-swap-event	–	+
glx-swap-exchange	–	–
glx-swap-pixmap	+	+
glx-swap-pixmap-bad	+	+
glx-swap-singlebuffer	+	+
glx-tfp	+	+
glx-visuals-depth	+	+
glx-visuals-stencil	–	+
glx-window-life	–	+

Примечание: под результатом, помеченным "*", понимается то, что конкретная версия X server не поддерживает GLX расширение, необходимое для работы теста, о чем было получено соответствующее сообщение. Последнее объясняется тем, что тестирование драйвера 1 и драйвера 2 происходило на одинаковых версиях X server, но собранных под разные архитектуры.

Те тесты, для которых результат помечен как "–", считаются завершившими свое выполнение с ошибкой. При этом мы не будем разделять, результаты типа "fail" и "skipped" или сообщение об ошибке драйвера.

Проведя анализ представленных результатов можно сделать следующие выводы:

1. Существующая версия драйвера nouveau реализует не все возможности GLX API. Это свидетельствует о том, что полная поддержка всех возможностей GLX и граничных случаев при использовании API GLX пока не реализована даже в проекте Mesa.

2. Как видно, практически всегда (отдельные случаи мы рассмотрим ниже), когда тест завершается ошибкой при тестировании драйвера 1, точно такое же поведение демонстрирует и драйвер 2.

3. Группа тестов с префиксом "glx-oml-sync-" завершается с ошибкой на драйвере 1 по причине того, что все они требуют GLX расширение "GLX_OML_sync_control", поддержка которого в настоящий момент времени отключена. Аналогичная причина ошибочного результата и по некоторым другим тестам.

Поскольку большая часть тестов GLX для драйвера 1 демонстрирует результат, совпадающий с результатами тестирования драйвера 2, можно считать, что в результате внедрения разработанного алгоритма взаимодействия между пользовательскими приложениями и графической подсистемой ОС Linux стало возможным запускать приложения, написанные с использованием GLX API в условиях отсутствия поддержки DRI драйвером графического ускорителя.

6. ЗАКЛЮЧЕНИЕ

Разработанный алгоритм может быть использован при проектировании драйверов новых графических ускорителей. Результаты исследования могут позволить разработчикам драйверов графических ускорителей сократить время разработки за счет отказа от реализации DRI2 подсистемы, поскольку разработанный алгоритм позволит запускать приложение, использующее GLX API через EGL.

Полученные результаты позволят сократить затраты ресурсов на поддержку отдельных подсистем графических драйверов и осуществить "бесшовный" переход на использование EGL API во встраиваемых системах.

Разработанный алгоритм обеспечит запуск приложений, написанных с использованием GLX API во встраиваемых системах на основе ОС Linux с драйверами без поддержки DRI2. Представлено формальное описание алгоритма и проведено тестирование его работы.

Дальнейшие исследования возможны в направлении улучшения работы метода в направлении обеспечения полного прохождения тестов piglit.

ЛИТЕРАТУРА

1. Zhihao Y, Zongheng M, Yingtong L, Ardalan S, Chandramowlishwaran A. Sugar: Secure GPU Acceleration in Web Browsers. *ACM SIGPLAN Notices*, 2018, 53(2):519-534. https://doi.org/10.1145/3296957.3173186.
2. Stone JE, Messmer P, Sisneros R, Schulten K. High performance molecular visualization: In-situ and parallel rendering with EGL. *International Parallel and Distributed Processing Symposium workshops IEEE*, 2016. https://doi.org/10.1109/IPDPSW.2016.127.

3. Hou D, Gan J, Li Y, Yazami YE, Jain T. Transparent Checkpointing for OpenGL Applications on GPUs. *ArXiv*, 2021. abs/2103.04916.
4. Volk AO, Ivanova VA, Syschikov AYu, Sedov BN. The Indicators Framework for Developing Display Systems. *Proc. Conf. "Wave Electronics and its Application in Information and Telecommunication Systems" (WECONF)*, St. Petersburg, Russia, 2020, pp. 1-7, doi: 10.1109/WECONF48837.2020.9131477.
5. Wang L, Jia H, Zhang Y, Li K, Wei C, EgpuIP: An Embedded GPU Accelerated Library for Image Processing. *IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys),* Hainan, China, 2022, pp. 914-921, doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00147.
6. Li Y, Zhang S, Li J, Xi H, Dong T, Du Z. Research on Operating System Migration Method Based on Domestic Embedded Devices. *Journal of Physics: Conference Series,* 2022, 2171(1):012067. DOI 10.1088/1742-6596/2171/1/012067
7. Gao D, Lin H, Li Z, Liu Y, Qian F. Trinity: high-performance and reliable mobile emulation through graphics projection. *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 285-301. https://doi.org/10.1145/3643029
8. Qin P, Xia Z, Guoxian G, Xiaofang T, Li G. GPU-Based In Situ Visualization for Large-Scale Discrete Element Simulations. *Wireless Communications and Mobile Computing,* 2022, Volume (2022). DOI: 10.1155/2022/3485505.
9. Nakhoon B, Kwan-Hee Y. An accelerated rendering scheme for massively large point cloud data. *The Journal of Supercomputing,* 2020, 76:8313-8323. DOI: 10.1007/s11227-019-03114-y.
10. Пугин КВ, Гиацинтов АМ, Мамросенко КА. Графический стек для Linux на базе OpenGL ES, glvnd и Glamor. *Вычислительные технологии*, 2023, 28(1):89-102; doi: 10.25743/ICT.2023.282.008'
11. Young KZ. Unicon's OpenGL 2D and Integrated 2D/3D Graphics Implementation, 2020. http://unicon.org/reports/young.pdf.

Татарчук Иван Александрович
младший научный сотрудник
НИИ Системных исследований РАН
36/1, Нахимовский просп., Москва 117218, Россия
E-mail: rilian@niisi.ras.ru

Мамросенко Кирилл Анатольевич
к.т.н., руководитель Центра
НИИ Системных исследований РАН
36/1, Нахимовский просп., Москва 117218, Россия
E-mail: mamrosenko_k@niisi.ras.ru

Гиацинтов Александр Михайлович
к.т.н., старший научный сотрудник
НИИ Системных исследований РАН
36/1, Нахимовский просп., Москва 117218, Россия
E-mail: algts@niisi.ras.ru.