ПРОГРАММИРОВАНИЕ, 2021, № 6, с. 51–61

КОМПЬЮТЕРНАЯ ГРАФИКА И ВИЗУАЛИЗАЦИЯ

УДК 004.92

МНОГООКОННАЯ ВИЗУАЛИЗАЦИЯ АВИАЦИОННОГО ДИСПЛЕЯ С ИСПОЛЬЗОВАНИЕМ АППАРАТНОГО УСКОРЕНИЯ

© 2021 г. Б. Х. Барладян[a], Н. Б. Дерябин[a], Л. З. Шапиро[a], Ю. А. Солоделов[b], А. Г. Волобой[a,*], В. А. Галактионов[a]

[a] *Институт прикладной математики им. М.В. Келдыша РАН, Москва, Россия*

[b] *Государственный научно-исследовательский институт авиационных систем, Москва, Россия*

** E-mail: voloboy@gin.keldysh.ru*

Поступила в редакцию 13.06.2021 г.
После доработки 16.07.2021 г.
Принята к публикации 22.07.2021 г.

Современный дисплей пилота гражданского самолета основан на новой идеологии интерфейса и позволяет улучшить восприятие полетной информации из нескольких источников за счет ее объединения на одном многофункциональном дисплее. В работе рассматриваются вопросы реализации многооконной визуализации дисплея пилота при использовании OpenGL SC с аппаратным ускорением. Предложен алгоритм компоновки информации на дисплее, позволяющий применять только одно GPU устройство, доступное на борту самолета. Подробно изложен подход адаптации и модификации пакета Mesa с открытым программным кодом для получения сертифицируемого драйвера GPU. Особое внимание уделено технологии адаптации открытых кодов пакета к операционной системе реального времени и к требованиям к системам, критичным для безопасности. Реализация предложенного подхода предназначена для работы под управлением операционной системы реального времени JetOS в системах визуализации бортовых комплексов гражданской авиации. Описанная реализация многооконной визуализации предполагает в дальнейшем ее сертификацию для систем, критичных для безопасности.

DOI: 10.31857/S0132347421060029

1. ВВЕДЕНИЕ

В последние десятилетия значительно усложнились кабины экипажей гражданских самолетов. Современная техника позволяет показывать пилоту больше информации об авиационном оборудовании и полетной ситуации. Увеличение вместимости пассажирских судов и продолжительности полетов в сложных метеоусловиях также увеличивают психологическую нагрузку на пилотов. Это потребовало разработки новой концепции дисплея. Соответственно, и индикаторы, и компоновка панели дисплея должны быть удобными для пользователя, чтобы улучшить взаимодействие между пилотом и летательным аппаратом [1, 2]. В связи с этим вопросы проектирования и внедрения приборов в кабину экипажа имеют большое значение с точки зрения обеспечения безопасной и эффективной работы пилотов.

С внедрением электронных систем управления полетом современный самолет получил “стеклянную кабину”. Эта новая идеология интерфейса позволяет улучшить восприятие полетных данных [3] за счет объединения важной информации на одном многофункциональном дисплее, который обеспечивает интегрированное, легко понятное изображение самолета. В настоящее время информация из нескольких источников визуализируется сразу на одном большом экране (например, приборная панель самолета Boeing 737 MAX, показанная на рис. 1). Кроме того, дисплей можно настраивать и отображать разную информацию в разных сегментах полета.

Современные бортовые системы проектируются на основе IMA (Integrated Modular Avionics) архитектуры [4]. Ключевой особенностью этой архитектуры является возможность выполнения нескольких функциональных приложений, реализующих программную часть систем авионики, на одном компьютере. Для этого необходимо совместное использование времени и ресурсов между приложениями. Этот режим обеспечивается бортовой операционной системой реального времени, под управлением которой и выполняются приложения. В [5] сформулированы требования к операционной системе реального времени (ОСРВ), предназначенной для работы с интегрированной модульной авионикой. В частности, ОСРВ должна соответствовать стандарту ARINC 653 [6]. Необходимость сертификации требует соблюдения

Рис. 1. Приборная панель Boeing 737 MAX.

корректных процессов разработки программного обеспечения (ПО) в соответствии с DO-178C [7], а также полного доступа к исходным кодам. Разрабатываемая нами система визуализации предназначена для работы под ОСРВ JetOS [8].

Важной составляющей системы визуализации бортового дисплея является библиотека OpenGL, которая для использования в авиационном ПО должна удовлетворять стандарту OpenGL SC (Safety Critical). В работах [9, 10] была представлена программная реализация графической библиотеки OpenGL SC, работающая под управлением ОСРВ JetOS. Программную реализацию OpenGL проще сертифицировать в соответствии с описанными выше требованиями, однако программные решения уступают по скорости визуализации аппаратной реализации библиотеки. Скорости визуализации, достигнутые в работах [9, 10], оказались удовлетворительными не для всех типичных авиационных приложений. Более того, для большинства анализируемых приложений удовлетворительные скорости были достигнуты только при использовании четырех ядер процессора. Однако использование четырех ядер процессора не всегда допустимо, поскольку в бортовой системе есть и другие потребители вычислительных ресурсов. Это также создает определенные проблемы для сертификации системы. В силу этих причин представляет большой практический интерес реализация библиотеки OpenGL SC с использованием аппаратного ускорения. Одной из широко используемых авиационных платформ является процессор i.MX6 с графическим ускорителем Vivante, для которых соответственно и необходимо разработать систему визуализации дисплея.

2. АЛГОРИТМ МНОГООКОННОЙ ВИЗУАЛИЗАЦИИ С ИСПОЛЬЗОВАНИЕМ АППАРАТНОГО УСКОРЕНИЯ

В случае, когда каждое приложение может использовать собственный экземпляр OpenGL, задача построения многооконного изображения решается путем использования компоновщика (compositor) [9, 11]. Для разработанной нами программной OpenGL SC мы также реализовали компоновщик [10], при использовании которого на многоядерных компьютерах была достигнута минимально приемлемая скорость визуализации для ряда не очень сложных авиационных приложений. Но проблема в том, что программная OpenGL на используемых в бортовом оборудовании не очень мощных компьютерах не может обеспечить необходимую скорость визуализации для сложных приложений.

При применении OpenGL с поддержкой аппаратного ускорения оба подхода, разработанные в [10] для многооконной визуализации, не могут быть использованы, поскольку они предполагают применение в каждом приложении отдельного экземпляра OpenGL. Такой подход невозможен при использовании аппаратного ускорения, поскольку в типичной авиационной платформе имеется только один графический процессор. Соответственно, не можем использовать отдельные экземпляры OpenGL в каждом разделе или экземпляре JetOS. Для решения этой проблемы мы разработали новый подход, когда все команды OpenGL выполняются в одном специальном разделе операционной системы JetOS.

Работу приложения, использующего библиотеку OpenGL для визуализации, можно разделить на две компоненты:

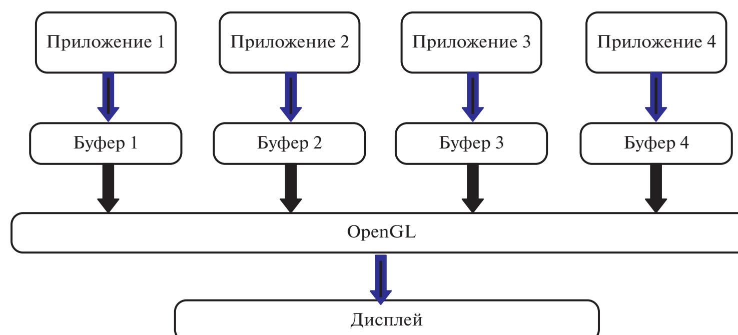

Рис. 2. Схема работы многооконной визуализации с использованием аппаратного ускорения.

1. Подготовка различных данных, необходимых для работы OpenGL – параметры геометрии, различные атрибуты, параметры камеры и т.д.

2. Непосредственный вызов функций OpenGL с подготовленными параметрами.

Для использования аппаратного ускорения в OpenGL в нашей реализации эти компоненты выполняются в разных разделах операционной системы JetOS. Каждое приложение выполняет подготовку необходимых данных в своем разделе и записывает необходимые для визуализации функции OpenGL с соответствующими параметрами в специальный массив. Этот массив находится в общей памяти с разделом, который осуществляет реальные вызовы функций OpenGL. Когда все вызовы, необходимые для генерации изображения одного кадра, подготовлены, то с помощью функций синхронизации запускается работа раздела, работающего с библиотекой OpenGL. Работу системы в целом можно представить с помощью блок-схемы, приведенной на рис. 2.

Код каждого приложения используется практически без изменений. Минимальные изменения на верхнем уровне приложения вносятся только для синхронизации работы приложений и раздела, осуществляющего реальные вызовы функций OpenGL.

Нами были созданы библиотеки OGLOUT и OGLIN. Вместо реальных вызовов функций OpenGL с помощью библиотеки OGLOUT осуществляется кодирование вызовов функций и запись необходимых данных в выходной массив соответствующего буфера. Эти кодирующие функции относительно просты. Если функция просто передает фиксированное число параметров, то в текущую позицию массива записывается индекс вызываемой функции, а за ним подряд значения параметров. Это относится к большинству функций OpenGL таких, как, например, glClearColor(), glClearDepthf() и т.д. В более сложных случаях, когда передаются указатели на массивы неопределенной в момент вызова длины, например glVertexPointer(), технология кодирования вызова несколько усложняется. В момент вызова таких функций вместо указателя записывается индекс зарезервированной позиции в выходном массиве, куда затем будут записываться используемые по данному указателю значения в функциях glMultiDrawArrays() или glDrawElements(). Только эти две функции работают с массивами неопределенной длины.

Декодирование записанной приложениями информации в вызовы функций OpenGL осуществляется с помощью библиотеки OGLIN последовательно для каждого массива данных, сгенерированного соответствующим приложением. Библиотека OGLIN состоит из одной функции process_ogl_input_array(), которая последовательно читает данные, записанные приложением. Затем по прочитанному индексу функции OpenGL переходит к участку кода, написанного для данной функции, извлекает из последующих элементов переданные данные, включая значения указателей, и вызывает с этими параметрами заданную функцию OpenGL.

Некоторым недостатком этого подхода является сложность в реализации функций OpenGL, которые возвращают значения параметров (такие как, например, glGenLists (), glGetError () и др.). Однако анализ практических приложений, применяемых в настоящее время в бортовом оборудовании, показал, что эти функции в них не используются. На практике эти функции используются, как правило, только при отладке приложений при однооконной визуализации. Вызовы этих функций также отсутствуют в библиотеке OGLX [12], которая в настоящее время используется при разработке большинства авиационных приложений с выводом на экран дисплея.

Синхронизация работы приложений и раздела, непосредственно вызывающего функции OpenGL, реализована с помощью специальных объектов, называемых событиями. События – это объекты, которые бывают в двух состояниях: взведен и сброшен. Поскольку эти объекты должны быть доступны одновременно из двух разделов (из приложения и из OpenGL), то они в нашем случае были реализованы через небольшую область памяти, общую для двух разделов. Детально реализация событий описана в работе [10]. Для синхро-

низации работы приложения и OpenGL раздела используется пара событий:

1. StartRender – взводится приложением, когда подготовлены вызовы всех функций OpenGL, необходимых для генерации изображения для данного кадра;

2. EndRender – взводится в OpenGL-разделе, когда генерация изображения для заданного кадра закончена.

Теперь алгоритм работы приложения можно описать следующим алгоритмом:

```
While (true)
{
 WaitEvent(EndRender);
 ResetEvent(EndRender);
 ResetOGLOutput();
 Генерация вызовов OpenGL;
 SetEndOGLOutput();
 SetEvent(StartRender);
}
```

Функция ResetOGLOutput() инициализирует начало работы с выходным массивом, a SetEndOGLOutput() его завершает. Генерация вызовов OpenGL не отличается от ее прямого использования. Только функции OpenGL заменены соответствующими функциями библиотеки OGLOUT.

Работу OpenGL раздела в случае двух приложений можно представить следующим алгоритмом:

```
processed_partitions[0] = false;
processed_partitions [1] = false;
client_num = 2;
while(1)
{
 for (int ic = 0; ic < client_num; ic++)
 {
  if (GetEventState(StartRender [ic]) == SIGNALED)
  {
   ResetEvent(StartRender [ic]);
   process_ogl_input_array(oglinp[ic]);
   processed_partitions[ic] = true;
  }
  if (processed_partitions[0] && processed_partitions [1])
   break;
 }
 if (!processed_partitions[0] || !processed_partitions [1])
   continue;
 SwapBuffers();

 SetEvent(EndRender [0]);
 SetEvent(EndRender [1]);
 processed_partitions[0] = false;
 processed_partitions [1] = false;
}
```

Раздел ждет пока приложения не подготовят свои выходные файлы с данными и последовательностью вызовов функций OpenGL. Как только это произошло (*StartRender* [ic]) = SIGNALED), то запускается вызов этих функций. После того как вызовы функций закончены для всех приложений, вызывается функция SwapBuffers(), которая завершает работу OpenGL для данного кадра и выводит результирующее изображение на экран. После этого взводятся события *EndRender* для всех приложений, и они начинают генерировать вызовы функций OpenGL для следующего кадра.

3. РАЗРАБОТКА GPU ДРАЙВЕРА С ОТКРЫТЫМ КОДОМ

В большинстве случаев драйверы, предоставляемые производителями GPU, не имеют открытого программного кода. Это означает, что такое ПО не может быть сертифицировано для использования в авиации. Также драйвер необходимо адаптировать для работы под управлением ОСРВ JetOS. Реализация различных версий OpenGL SC на базе коммерческих драйверов рассматривается в работах [13–15], однако использование бинарных драйверов в ОСРВ JetOS невозможно.

Для решения этой задачи мы использовали пакет Mesa [16] с открытыми исходными кодами. Mesa представляет собой программную реализацию с открытым исходным кодом OpenGL, Vulkan и других спецификаций графического API. Mesa переводит эти спецификации в драйверы конкретного графического оборудования. Некоторые производители процессоров, такие как, например, AMD или Intel, сами разрабатывают драйверы для пакета Mesa с открытым кодом для производимых ими процессоров. Другие производители, такие как Nvidia или Vivante, полностью заменяют Mesa, обеспечивая собственную реализацию библиотеки OpenGL. Для такого оборудования сообщество разработчиков Mesa создает альтернативные открытые драйверы (reverse-engineering), такие как Nouveau для Nvidia или Etnaviv для Vivante. В нашем случае при использовании платформы i.MX6 с графическим процессором Vivante единственной возможностью является использование драйвера Etnaviv. Первые шаги по использованию пакета Mesa в ОСРВ JetOS, изложены в докладе [17].

Пакет Mesa в настоящее время адаптирован только для использования под управлением ОС Linux, Android и Windows. Для этих операционных систем доступна технология компиляция и интеграция пакета MESA для большинства используемых в настоящее время графических процессоров. Разработка приложений (драйверов и библиотек) для работы под управлением ОСРВ JetOS существенно отличается от технологии разработки соответствующих программных продук-

тов для работы под управлением таких операционных систем как Linux или Windows.

Для использования пакета Mesa в ОСРВ JetOS, удовлетворяющей стандарту ARINC 653, необходимо было решить следующие основные проблемы:

1. В пакете Mesa используется большое количество системных функций ОС Linux, большая часть из которых отсутствует в JetOS. Эти функции необходимо заменить соответствующими функциями JetOS в тех случаях, когда имеются аналоги. В остальных случаях надо реализовать эквивалентные функции, функциональности которых достаточны для наших задач.

2. Динамическое выделение памяти, освобождение памяти запрещено. Выделение памяти разрешается только на этапе инициализации раздела. Это значит, что соответствующие вызовы в пакете Mesa должны быть исключены или переписаны с использованием собственного специального диспетчера памяти. Для выделения памяти можно использовать только функции JetOS.

3. Многопотоковость отсутствует согласно ARINC 653. Это означает, что соответствующие вызовы Mesa, использующие объекты мьютекс (mutex), должны быть исключены или переписаны.

4. В пакете Mesa присутствует большое количество кода, предназначенного для использования в различных операционных системах и для работы с различными графическими процессорами. Весь избыточный код необходимо исключить в соответствии со стандартом DO-178C. Очевидно, что чем меньше размер используемого кода, тем проще процесс сертификации. Размер исходного кода пакета Mesa составляет ~125 мегабайт и содержит ~6000 файлов. Большая его часть не имеет отношения к нашей задаче.

Разработка GPU драйвера с открытым программным кодом происходила в несколько последовательных этапов.

3.1. Адаптация пакета Mesa для ОС JetOS

На этом этапе пакет Mesa портировался в виде одной общей библиотеки **libmesa** с минимальными изменениями, необходимыми для компиляции и сборки графических приложений под управлением ОСРВ JetOS. При этом решались следующие основные задачи:

1. Добавление необходимых компонент. Пакет MESA требует использования внешних пакетов **libdrm** (DRM интерфейс) и **expat** (парсер XML файлов).

2. Исключение заведомо ненужных компонент. Были исключены все драйверы (amd, broadcom, intel и множество других) кроме etnaviv и imx. Также были исключены тесты (например, gtest), поддержка X-Windows (glx) и т.п.

3. Исключение ненужных исходных файлов. Некоторые исходные файлы предназначены для отдельно собираемых программ, таких как тесты, компилятор шейдеров и т.п. Для исключения таких файлов анализировались файлы сборки пакета Mesa.

4. Добавление генерируемых файлов. Пакет Mesa активно использует генерируемые исходные файлы, не входящие в состав поставляемого пакета. Генерируемые файлы создаются во время сборки при помощи специальных скриптов, написанных на языке Python. К счастью, генерируемые файлы слабо зависят от конкретной операционной системы. Поэтому были использованы файлы, полученные при сборке пакета Mesa под операционной системой Linux.

5. Исключение динамической загрузки. Пакет Mesa использует динамическую загрузку для подключения, например, требуемых графических драйверов (в случае i.MX6 – это etnaviv и imx). Так как в JetOS пакет портируется как одна общая статическая библиотека, то динамическая загрузка заменяется прямым вызовом соответствующей функции. Компонента динамической загрузки loader была исключена.

6. Добавление компоненты системной поддержки для JetOS, реализующая необходимые системные функции.

В результате такой адаптации пакета Mesa для работы в ОСРВ JetOS и на платформе i.MX6 была достигнута работа графических приложений с использованием аппаратного ускорения графики. Размер адаптированной библиотеки значительно уменьшился, но остался все еще большим: более 30 мегабайт исходного кода (почти 700 файлов *.c, более 700 файлов *.h, более 100 файлов *.cpp). Причинами этого является реализация большого количества интерфейсов, избыточных для нашей задачи. В частности, Mesa реализует все версии OpenGL с их многочисленными расширениями. А для авиационных приложений необходима поддержка только интерфейсов OpenGL SC стандартов 1.0.1 и 2.0.1. Пакет Mesa также содержит компилятор шейдеров, который согласно стандарту OpenGL SC 2.0.1 должен использоваться только на этапе подготовки приложений. Сами шейдеры должны загружаться на этапе выполнения приложений в двоичном виде.

3.2. Организация структуры драйвера

Для того чтобы получить качественный, поддерживаемый, сертифицируемый результат (драйвер с открытым программным кодом), прежде всего, важна хорошая структуризация всего проекта. К сожалению, исходный пакет Mesa плохо структурирован. Это связано с двумя причинами. Во-первых, пакет поддерживает большое количество интерфейсов графических библиотек: OpenGL ES

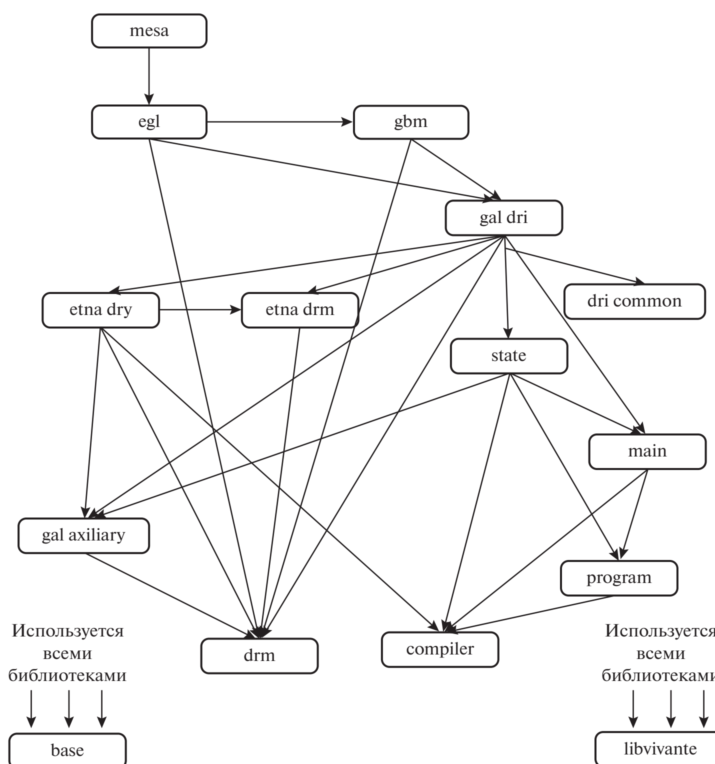

Рис. 3. Иерархия библиотек GPU драйвера.

(версии 1, 2, 3), OpenCL, OpenMAX, VDPAU, VA API, XvMC и Vulkan. Большую часть этих интерфейсов необходимо исключить. Во-вторых, в разработке пакета принимало участие большое количество разработчиков, которые преследовали различные цели и использовали различные программные трюки.

Поскольку нашей целью является поддержка двух стандартов OpenGL SC с достаточно жесткими ограничениями, то в качестве первого шага мы преобразовали адаптированный пакет, полученный на первом этапе, в иерархию библиотек. Под иерархией понимается, что библиотеки образуют дерево по отношению их использования: вышестоящие библиотеки используют функции и данные, определенные в нижестоящих библиотеках, но никак не наоборот. Более точно, для сборки (компиляции) каждой библиотеки в иерархии достаточно использовать только нижестоящие библиотеки. Для достижения этой цели ряд исходных файлов был разбит на части и разнесен по разным библиотекам. Такая же процедура была применена и к соответствующим заголовочным файлам. Чаще всего, одна из частей помещалась в "базовую" библиотеку (не использующую другие библиотеки).

Полученная иерархия библиотек и их зависимости показаны на рис. 3. Структура осталась достаточно сложной, но главный результат состоит в том, что иерархия все же была достигнута.

Для лучшего понимания полученной иерархии опишем функциональности некоторых библиотек на рис. 3:

• **mesa** – головная библиотека. Это один файл, который реализует интерфейс базовых функций, используемых графическими приложениями – инициализация буфера кадров, создания графического контекста, SwapBuffers() и др.

• **egl** – полная реализация стандартного интерфейса OpenGL с базовой оконной системой (EGL). Для работы библиотек OpenGL SC под управлением ОСРВ JetOS большая часть функций EGL не требуется, и они были исключены на следующих этапах работы.

• **gbm** – General Buffer Management. Компонента Linux, необходимая для работы с DRM (Rendering Manager).

• **etna_drv** – расширение драйвера Etnaviv для поддержки аппаратного ускорения.

• **etna_drm** – расширение драйвера Etnaviv для работы с DRM. Работа с буферными объектами и с буфером команд GPU.

• **state** – компонента, отслеживающая изменение состояния рендеринга OpenGL.

• **main** – содержит обработчики всех функций OpenGL.

• **program** – вспомогательная компонента, работающая с программами для шейдеров.

• **compiler** – компилятор шейдеров.

• **drm** – сторонняя библиотека, необходимая для работы с DRM.

• **base** – базовая библиотека, используемая всеми другими библиотеками. В ней собраны различные базовые функции, изъятые из других библиотек.

• **libvivante** – библиотека вывода на экран (Frame buffer library).

Разбивка на библиотеки сделала адаптированный код более управляемым, однако следует отметить, что поток управления в динамике в целом этой разбивке может не соответствовать, так как возможен вызов функций по указателю. Ряд компонент в пакете Mesa использует механизм виртуальных функций (как в C++), и такие вызовы могут не соответствовать иерархии рис. 3. Широко используется интерфейс (struct) pipe_context, через который вызываются аппаратно-зависимые функции драйвера, реализуемые, в основном, компонентой etna_drv, а также интерфейс Driver (struct dd_function_table), через который инициируются изменения состояния рендеринга, реализуемый библиотекой state.

3.3. Диспетчеризация OpenGL функций и генерация кода

Обработчики OpenGL функций в пакете Mesa вызываются через механизм диспетчеризации (dispatch). Имеется таблица обработчиков OpenGL функций, задаваемая указателем _glapi_Dispatch, которая может динамически переустанавливаться при помощи функции _glapi_set_dispatch(). Изначально установлена таблица пустых функций, так как до создания контекста рендеринга функции OpenGL не должны ничего делать. Когда устанавливается контекст рендеринга, устанавливается основная таблица диспетчеризации. Таблица диспетчеризации может также переключаться при выполнении функций glBegin() и glEnd().

Для каждой OpenGL функции gl*Func*() механизм диспетчеризации определяет три функции:

• GET_Func() – получить адрес обработчика функции из таблицы диспетчеризации;

• SET_Func() – установить новый обработчик функции в таблице диспетчеризации;

• CALL_Func() – выполнить обработчик функции, установленный для этой функции в таблице диспетчеризации (макро).

Это дает возможность манипулирования обработчиками для отдельных функций, и Mesa этим активно пользуется. Входные функции OpenGL просто вызывают соответствующий (по номеру) обработчик из текущей таблицы диспетчеризации. Таким образом, достигается гибкая настройка обработчиков OpenGL- функций для текущего состояния рендеринга.

В настоящее время пакет Mesa реализует более 1400 OpenGL функций (не считая синонимов). Для сравнения, спецификации OpenGL SC 1.0.1 и 2.0.1 используют немногим более 100 функций каждая. К тому же механизмы диспетчеризации в пакете Mesa перегружены избыточной для нашей задачи функциональностью и усложнены. Для наших целей эти компоненты было необходимо радикально переписать.

Рассматриваемые компоненты интенсивно используют генерируемые файлы. Как минимум, для каждой OpenGL функции gl*Func*() необходимы определения четырех функций – SET_*Func*(), GET_*Func*(), CALL_*Func*() и точки входа gl*Func*() – с соответствующими аргументами. Код этих функций и другие необходимые описания и определения генерируются в пакете Mesa при помощи специальных скриптов. Исходными файлами для их генерации служит большой набор XML-файлов. Этот набор принципиально настроен на полные стандарты OpenGL и OpenGL ES, а также на внутренние потребности пакета. Поэтому адаптировать этот генератор кода для целей нашей задачи оказалось практически невозможно.

Для решения этой проблемы компонента диспетчеризации была полностью переписана. Был реализован собственный генератор кода для компонент диспетчеризации. Генератор кода в виде скрипта на языке Python использует в качестве входа набор “официальных” заголовочных файлов OpenGL SC заданных стандартов 1.0.1 и 2.0.1. В результате генерируются только необходимые функции.

3.4. Библиотека аппаратного ускорения libhwgl

После реализации генератора кода для компоненты диспетчеризации стало возможным выделение из общей библиотеки libmesa, которая в текущем состоянии поддерживает стандарты OpenGL SC 1.0.1, 2.0.1, ES2 с некоторыми расширениями, библиотеки libhwgl, которая будет поддерживать только необходимые нам стандарты OpenGL SC 1.0.1 и 2.0.1. Для решения этой задачи в генераторе кода в качестве входа использовались только заголовочные файлы OpenGL SC заданных стандартов. Дальнейшая оптимизация кода производилась только для библиотеки libhwgl. Библиотека libmesa была оставлена в “замороженном” виде в основном для целей тестирования приложений, написанных по стандарту OpenGL ES2 под ОСРВ JetOS.

Как отмечалось выше, одной из основных проблем с исходным пакетом при приведении его к состоянию, удовлетворяющему требованиям к сертификации, состоит в его гигантском размере – более 30 мегабайт исходного кода после переноса

в JetOS, практически отсутствующей документации, сложности понимания и сопровождения. Для целей нашего проекта необходимо получить продукт, лишенный избыточного кода, с хорошей структурой, с соблюдением требуемых стандартов программирования и пригодный для будущей сертификации. В соответствии с DO-178C в программном обеспечении бортового оборудования не должно быть неисполняемого кода (Dead code), поэтому одной из важных задач было исключение кода, который не будет использоваться в OpenGL SC стандартов 1.0.1 и 2.0.1.

С этой целью был использован итерационный процесс, на каждом шаге которого достигается частичное снижение сложности и избыточности пакета. Первые шаги уже были сделаны на этапе разработки библиотеки libmesa, когда поддерживаемый ею интерфейс OpenGL был сведен к стандартам ES2, SC 1.0.1 и SC 2.0.1, а обработчики, не входящих в эти стандарты функций, удалены. Основные методы, применяемые в этом процессе, можно условно разбить на два класса – **формальные** и **содержательные.**

Формальные методы – это методы, которые не требуют понимания работы кода, а только понимания общих принципов программирования. Простейшим применением формального метода является удаление неиспользуемой функции. Если какая-то функция не используется – ее описание и определение удаляются.

После удаления неиспользуемой функции могут появиться другие неиспользуемые функции. Неиспользуемые статические функции, как правило, диагностируются компилятором. Для выявления неиспользуемых функций (или части кода) может также применяться такой инструмент JetOS как “Сбор покрытия по коду”. Он позволяет отследить все функции, которые не использовались в данном приложении. Разумеется, эта функциональность должна применяться с осторожностью и не совсем формально, поскольку некоторые функции могут не использоваться в данном приложении, но применяться в других.

Аналогично, можно удалить неиспользуемую переменную или неиспользуемый член структуры. Переменная (поле структуры) не используется, если она (оно) нигде не читается. При этом переменной может что-то присваиваться – это еще не делает ее используемой. Такие присваивания, естественно, тоже удаляются.

Иногда переменной (члену структуры) присваивается только одно значение. В этом случае ее тоже можно удалить, заменив чтения этой переменной ее значением. Формальное удаление неиспользуемой константы (обычно – макро-константы, определенной через #define) возможно, если значение этой константы только читается (сравнивается с чем-либо, используется как альтернатива в switch и т.п.), но ничему не присваивается.

При удалении сравнения с константой в условном операторе (if) условие может стать истинным или ложным и тогда можно удалить неиспользуемые ветви или даже весь условный оператор. Альтернативы в операторе выбора (switch) с использованием данной константы удаляются, удаляется и соответствующая ветвь кода, если для нее не осталось других альтернатив. Если в операторе выбора альтернатив не осталось, удаляется весь оператор выбора (заменяется действиями по умолчанию, если таковые есть). Если в процессе адаптации функция стала пустой, то ее и ее вызовы также можно удалить. Если функция стала возвращать единственное значение, ее тоже можно удалить, заменив вызовы этой функции данным значением. Если в файле не осталось внешних функций и переменных, он удаляется, и т.д.

Содержательные методы адаптации Mesa основаны на полном понимании работы отдельных компонент пакета. Суть содержательного метода состоит в замене кода компонент на более простой, понятный, поддерживаемый и поддающийся формальной верификации код.

В качестве примера применения содержательного метода можно рассмотреть удаление виртуального интерфейса. Исходный пакет Mesa имеет дело с самыми разнообразными драйверами и протоколами, а также с одновременной работой с разными устройствами. Поэтому он часто и оправданно использует виртуальные интерфейсы. Однако для нашей задачи виртуальные интерфейсы не нужны, так как по ним всегда вызывается один и тот же код. Более того, в нашем случае они часто оказываются вредными, так как будучи написанными на языке C с использованием специальных добавочных структур, существенно затрудняют понимание исходного кода (по вызову виртуальной функции бывает сложно определить реально вызываемую функцию). Удаление виртуального интерфейса состоит в замене вызовов виртуальных функций прямыми вызовами. Однако эта операция не так тривиальна, как может показаться, поскольку надо понимать все детали работы кода в данном месте.

Другим примером применения содержательного метода является полная замена компоненты диспетчеризации OpenGL простым эквивалентным кодом, описанная в разделе 3.3.

4. РЕЗУЛЬТАТЫ

Сначала разработанное аппаратное ускорение было протестировано на тех же авиационных приложениях, которые использовались для тестирования программной реализации библиоте-

ки в работе [9]. Это приложения S_PFD (основной дисплей полета SSJ-100), M_PFD (он же для MC-21), Counters, GlassCockpit, FlightDisplay и relief (визуализация рельефа). Сравнивались скорости визуализации на платформе i.MX6 (Wandboard) при использовании программной реализации прототипа OpenGL SC 1.0.1 (SWGL) из [9] и аппаратной реализации OpenGL (HWGL), разработанной нами на базе пакета Mesa. Результаты сравнения приведены в таблице 1. Программная реализация OpenGL позволяет использовать несколько ядер процессора, поэтому в колонке SWGL 1 представлена скорость при использовании одного ядра процессора, а в колонке SWGL 4 при использовании 4 ядер процессора.

Следует отметить, что указанная в таблице для некоторых приложений скорость в 60 кадров в секунду ограничена синхронизацией с частотой обновления дисплея. Скорость работы непосредственно библиотеки HWGL для этих приложений выше. Таким образом, использование аппаратной поддержки для реализации библиотеки OpenGL SC 1.0.1 в операционной системе JetOS позволяют достичь существенного ускорения визуализации при использовании графического процессора Vivante.

Из табл. 1 видно, что достигнутая скорость визуализации для приложения S_PFD недостаточна для практического использования в бортовых системах. Анализ выполнения кода показал, что причиной этого является неэффективный код приложения, сгенерированный с использованием пакета SCADE [12]. Когда нам удалось реорганизовать вызовы OpenGL внутри этого приложения, то скорость визуализации возросла до 60 кадров в секунду. То есть скорость визуализации стала теперь ограничиваться не OpenGL, а синхронизацией с дисплеем.

Разработанный алгоритм многооконной визуализации был протестирован на двух парах практических приложений, которые использовались при тестировании в работе [18]. Это PFD (Primary Flight Display) с DOORS (состояние дверей самолета) – рис. 4, и PFD с ND (Navigation Display) – рис. 5.

Таблица 1. Скорость визуализации библиотеки OpenGL SC 1.0.1 с аппаратной поддержкой и программной реализацией. Скорость указана в кадрах в секунду

	SWGL 1	SWGL 4	HWGL
S_PFD	5.9	13.8	10.8
M_PFD	6.3	15.6	20.0
Counters	23.9	35.4	60
GlassCockpit	10.5	28.7	29.7
FlightDisplay	9.7	26.4	60
SVS	7.7	20.9	60

Таблица 2. Скорость многооконной визуализации программной и аппаратной OpenGL SC библиотек. Скорость указана в кадрах в секунду

	SWGL 1	SWGL 3	HWGL
PFD + DOORS	4.5	6.2	14.9
PFD + ND	4.6	6.2	16.5

В табл. 2 представлено сравнение скоростей многооконной визуализации для программной (SWGL) и аппаратной (HWGL) OpenGL SC библиотек. Визуализация HWGL использует алгоритм, описанный в разделе 2. В столбце SWGL 1 представлена скорость при использовании 1 ядра процессора, в то время как SWGL 3 использует 3 ядра – по одному для каждого приложения и одно для компоновщика [18].

5. ЗАКЛЮЧЕНИЕ

Разработка системы визуализации данных о полете и отображения информации о состоянии самолета обладает своей спецификой, связанной

Рис. 4. Многооконная визуализация двух приложений – PFD и DOORS.

Рис. 5. Многооконная визуализация двух приложений – PFD и ND.

с критически важными вопросами безопасности. Эта специфика часто не позволяет применять готовые, известные решения для той или другой функциональности. Предложенный в работе подход позволяет эффективно реализовать многооконный режим визуализации под управлением ОСРВ JetOS и при использовании аппаратной поддержки. При исследованиях использовалась библиотека OpenGL SC, реализованная путем адаптации пакета Mesa с открытым исходным кодом. Хотя при создании аппаратной OpenGL мы ориентировались на платформу i.MX6 и графический процессор Vivante, но разработанные алгоритмы и подходы могут быть использованы и для другого оборудования.

Представленный в данной статье подход к визуализации (HWGL + новый многооконный алгоритм) в итоге обеспечивает скорость в 2–3 раза превышающую скорость визуализации при использовании программных библиотек. Учитывая, что дисплей пилота предназначен для визуализации информации о полете и состоянии систем самолета, скорость в 15–20 кадров в секунду можно считать приемлемой.

СПИСОК ЛИТЕРАТУРЫ

1. *Senol M.B.* A new optimization model for design of traditional cockpit interfaces, Aircraft Engineering and Aerospace Technology. 2020. V. 92. № 3. P. 404–417. https://doi.org/10.1108/AEAT-04-2019-0068
2. *Thomas P., Biswas P., Langdon P.* State-of-the-Art and Future Concepts for Interaction in Aircraft Cockpits, Lecture Notes in Computer Science. 2015. V. 9176. P. 538–549. https://doi.org/10.1007/978-3-319-20681-3_51
3. *Kal'avsky P., Rozenberg R., Mikula B., Zgodavova Z.* Pilots' Performance in Changing from Analogue to Glass Cockpits, In Proc. of the 22nd Int. Scientific Conf. on Transport Means (Transport Means). 2018. P. 1104–1109.
4. *Федосов Е.А.* Проект создания нового поколения интегрированной модульной авионики с открытой архитектурой. Полет. 2008. № 8. С. 15–22.
5. *Федосов Е.А., Ковернинский И.В., Кан А.В., Солоделов Ю.А.* Применение операционных систем реального времени в интегрированной модульной авионики. OSDAY 2015, http://osday.ru/solodelov.html
6. *Ananda C.M., Nair S., Mainak G.* ARINC 653 API and its application – An insight into Avionics System Case Study. Defence Science Journal. V. 63. № 2. P. 223–229. https://doi.org/10.14429/dsj.63.4268
7. DO-178C Software Considerations in Airborne Systems and Equipment Certification. http://www.rtca.org/store_product.asp?prodid=803
8. *Маллачиев К.М., Пакулин Н.В., Хорошилов А.В.* Устройство и архитектура операционной системы реального времени. Труды ИСП РАН. 2016. Т. 28. Вып. 2. С. 181–192. https://doi.org/10.15514/ISPRAS-2016-28(2)-12
9. *Барладян Б.Х., Волобой А.Г., Галактионов В.А., Князь В.В., Ковернинский И.В., Солоделов Ю.А., Фролов В.А., Шапиро Л.З.* Эффективная реализация OpenGL SC для авиационных встраиваемых систем // Программирование. 2018. № 4. С. 3–10. https://doi.org/10.31857/S013234740000519-5
10. *Барладян Б.Х., Шапиро Л.З., Малачиев К.А., Хорошилов А.И., Солоделов Ю.А., Волобой А.Г., Галактионов В.А., Ковернинский И.В.* Система визуализации для авиационной ОС реального времени JetOS. Труды Института системного программирования РАН. 2020. Т. 32. Вып. 1. С. 57–70. https://doi.org/10.15514/ISPRAS-2020-32(1)-3
11. EGL_EXT_compositor *http://www.coreavi.com/sites/default/files/coreavi_product_brief_-_egl_ext_compositor.pdf*.
12. Ansys SCADE Display Capabilities https://www.ansys.com/products/embedded-software/ansys-scade-display/scade-display-capabilities.
13. *Baek N. and Lee H.* OpenGL ES 1.1 Implementation Based on OpenGL, Multimedia Tools and Applications. 2012. V. 57. № 3. P. 669–685.
14. *Baek N., Lee H.* OpenGL SC Implementation over an OpenGL ES 1.1 Graphics Board, 2012 IEEE International Conference on Multimedia & Expo Workshops (ICMEW 2012). P. 671–671. https://doi.org/10.1109/ICMEW.2012.127

15. *Baek N. and Kim K.J.*, Design and implementation of OpenGL SC 2.0 rendering pipeline. Cluster Computing. 2019. V. 22. P. S931–S936. https://doi.org/10.1007/s10586-017-1111-1
16. The Mesa 3D Graphics Library. https://www.mesa3d.org/
17. *Barladian B., Deryabin N., Voloboy A., Galaktionov V., Shapiro L.* High Speed Visualization in the JetOS Aviation Operating System Using Hardware Acceleration, CEUR Workshop Proceedings. 2020. V. 2744. Proc. of the 30th International Conference on Computer Graphics and Vision. pp. short3-1–short3-9. https://doi.org/10.51130/graphicon-2020-2-4-3
18. *Barladian B.Kh., Shapiro L.Z., Mallachiev K.M., Khoroshilov A.V., Solodelov Y.A., Voloboy A.G., Galaktionov V.A., Koverninskiy I.V.* Multi-windows rendering using software OpenGL in avionics embedded systems // CEUR Workshop Proceedings. V. 2485. Proc. of the 29th International Conference on Computer Graphics and Vision, 2019. P. 28–31. https://doi.org/10.30987/graphicon-2019-2-28-31