Программная реализация OpenGL SC для авиационных встраиваемых систем

Б.Х. Барладян[1], А.Г.Волобой[1], В.А. Галактионов[1], В.В. Князь[2], И.В. Ковернинский[2], Ю. А. Солоделов[2], В.А. Фролов[1], Л.З. Шапиро[1]
bbarladian@gmail.com|voloboy@gin.keldysh.ru|vlgal@gin.keldysh.ru|vl.kniaz@gosniias.ru|
ivkoverninsk@2100.gosniias.ru|yasolodelov@2100.gosniias.ru|vfrolov@graphics.cs.msu.ru|pls@gin.keldysh.ru
[1]Институт прикладной математики им. М.В.Келдыша РАН, Москва;
[2] ФГУП Государственный Научно-Исследовательский Институт Авиационных Систем (ГосНИИАС)

В работе рассматриваются вопросы повышения эффективности программной реализации библиотеки OpenGL SC для ее использования в авиационных встраиваемых системах. Алгоритмы растеризации были оптимизированы с учетом специфики авиационных приложений. Для ускорения визуализации применялись многопотоковые вычисления.

Ключевые слова: *OpenGL SC, ускорение визуализации, многопотоковые вычисления.*

Sofware OpenGL SC for aviation embedded systems

B. Kh. Barladian[1], A.G.Voloboy[1], V.A. Galaktionov[1], V.V. Kniaz[2], I. V. Koverninskiy[2], Y. A. Solodelov[2], L.Z.Shapiro[1]
[1]The Keldysh Institute of Applied Mathematics Russian Academy of Science;
[2] State Research Institute of Aviation Systems (GosNIIAS)

The rendering efficiency problems of multipurpose OpenGL SC for specific applications of embedded aviation systems are considered. Rasterization algorithms were optimized taking into account aviation specific. The multithreading calculations were applied for rendering acceleration.

Keywords: *OpenGL SC, rendering acceleration, multithreading calculations.*

1. Введение

В [1] сформулированы требования к операционной системе реального времени, предназначенной для работы с интегрированной модульной авионикой. В частности, должна быть обеспечена поддержка стандартов OpenGL SC/ES [2] и ARINC 661 [3] и возможность сертификации по DO-178C[4]. Необходимость сертификации требует соблюдения корректных процессов разработки ПО, а также полного доступа к исходным кодам библиотеки OpenGL. Важным требованием является независимость от платформы – ядро библиотеки не должно содержать кода, специфичного для конкретной архитектуры или аппаратной платформы. Компания CoreAVI выпускает OpenGL драйверы, использующие графические процессоры [5], но сертификация таких драйверов невозможна без участия разработчиков графических процессоров. В силу этих требований разработка программной OpenGL является практически единственным решением.

Конечно, с помощью программных решений трудно превзойти по скорости визуализации аппаратную реализацию библиотеки. Однако программное решение существенно проще оптимизировать для специальных приложений [6]. Предполагается, что разрабатываемая библиотека OpenGL будет работать в операционной системе реального времени JetOS [7]. В настоящее время JetOS не реализована в полном объеме, в частности, не реализована поддержка потоковых вычислений, поэтому основные исследования проводились под операционной системой Linux.

В настоящее время в литературе описано достаточно много алгоритмов программной реализации OpenGL [8], [9], [10], реализация которых на современных мощных компьютерах, например Intel i7-4770 3.4 GHz, дает вполне приемлемые по скорости результаты для типичных авиационных приложений. Скорость близка к показателям, показываемым OpenGL драйвером для видеокарты NVIDIA Quadro 410 - ~50 кадров в секунду. К сожалению, та же библиотека OpenGL не обеспечивает требуемой производительности на типичном авиационном компьютере на базе процессора PowerPC e500mc, 4 ядра, 1 ГГц [11].

Типичное авиационное приложение «Пилотажно-навигационный дисплей» (Primary flight display – PFD в англоязычной литературе) на рис. 1 работает со скоростью ~1.7 кадров в секунду, а визуализация рельефа на рис. 2 со скоростью ~1.4 кадров в секунду. Такие скорости визуализации неприемлемы для авиационных приложений.

Мы проанализировали специфику использования авиационными приложениями библиотеки OpenGL. Выяснилось, что большая часть этих приложений использует визуализацию только двумерных (2D) объектов. Эти приложения применяются, в основном, для визуализации значений различных приборов в цифровом и аналоговом виде, положения воздушного судна в пространстве, различных индикаторов состояния систем, метеорологической и картографической информации. Такая визуализация использует ограниченное число сочетаний вызовов OpenGL, учитывая которое мы сумели существенно увеличить скорость визуализации для специализированных приложений. При этом полное покрытие спецификации стандарта OpenGL SC было сохранено.

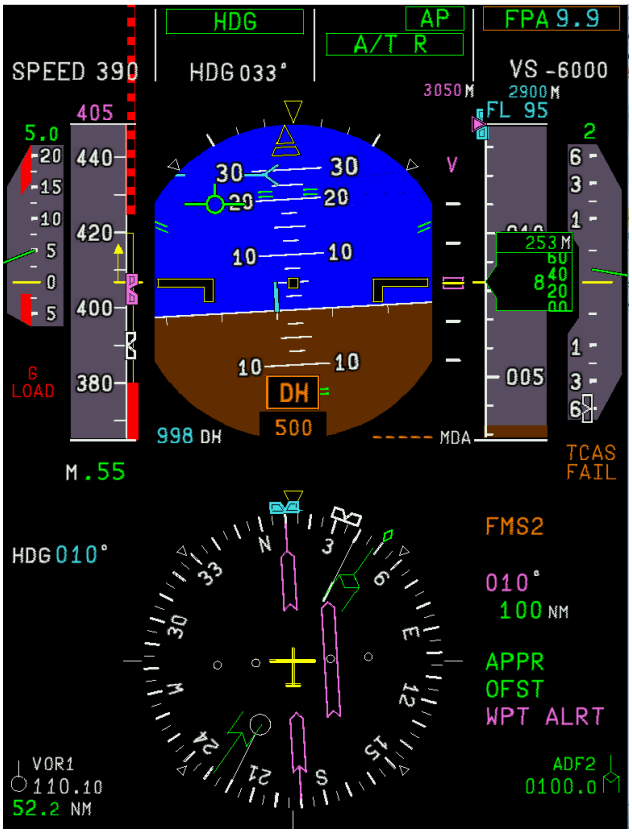

Рис. 1. Пилотажно-навигационный дисплей.

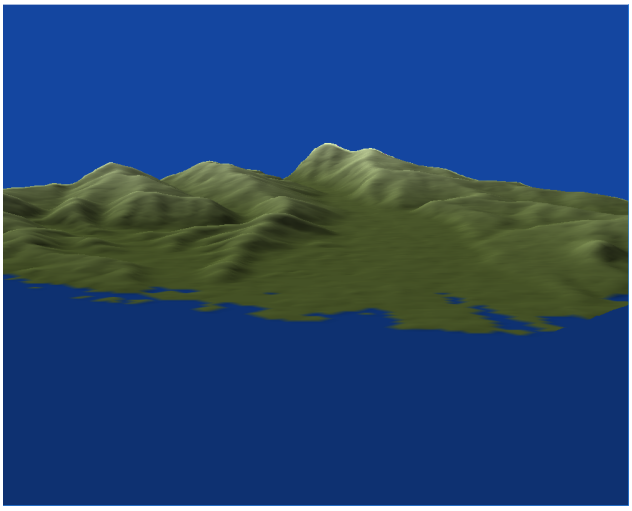

Рис. 2. Визуализация рельефа.

2. OpenGL алгоритмы – специфика использования

Алгоритмы растеризации занимают основное время при работе авиационных приложений. Большинство используемых в настоящее время приложений работают с двухмерными объектами, типичным представителем которых является PFD, приведенное на рис. 1.

Ускорение процедуры растеризации было проведено, в основном, путем применения следующих подходов:

1. Для вычисления значений величин (цвета или текстурных координат) внутри треугольника мы использовали линейную интерполяцию вместо применения барицентрических координат. Интерполяция производится сначала вдоль сторон треугольника, а затем вдоль строки пикселей. Использование линейной интерполяции вдоль отрезка вместо применения барицентрических координат позволило ускорить растеризацию примерно в 1.9 раза.
2. Везде, где это возможно, мы использовали вычисления с фиксированной точкой. Это дало заметное ускорение в силу специфики архитектуры процессора PowerPC – несколько таких операций могут выполняться за один такт. Использование вычислений с фиксированной точкой позволило ускорить растеризацию примерно в 2.2 раза.
3. Для вычислений в строке пикселей внутри треугольника был использован набор специальных встраиваемых функций.

На последнем пункте следует остановиться отдельно. При анализе 2D авиационных приложений было выявлено, что при растеризации треугольников большая часть времени уходит на проверки условий обработки данного пикселя, в то время как эти условия одинаковы для всех пикселей треугольника. Мы выявили пять таких часто используемых случаев:

1. Заполняется только буфер трафарета (стенсил);
2. Тестируется буфер трафарета, а вершины треугольника имеют один и тот же цвет;
3. Тестируется буфер трафарета, текстура заменяет альфа канал и вершины треугольника имеют один и тот же цвет;
4. Буфер трафарета не используется, текстура заменяет альфа канал и вершины треугольника имеют один и тот же цвет;
5. Вершины треугольника имеют один и тот же цвет и никаких других условий проверять не надо.

При визуализации рельефа (3D приложение) для эффективности, как правило, не используются буфер трафарета, альфа канал, текстура и освещение. Освещение вычисляется один раз заранее и результат в виде цветов в вершинах используется при визуализации. Таким образом, для 3D приложения, где используется Z буфер, рассматриваем два дополнительных случая: вершины треугольника имеют один и тот же цвет или они имеют разные цвета. Если вершины одного цвета, то не нужна интерполяция и цвет, которым они представлены в экранном буфере цвета, можно заранее перед растеризацией упаковать в одну целую величину (тип int). Отсутствие необходимости интерполяции и упаковки цвета на каждом пикселе позволяет существенно повысить скорость растеризации, поэтому мы всегда выделяем этот случай (в сочетании с другими условиями, когда это необходимо). Таким образом, добавляются еще два специальных случая:

1. Используется Z-буфер, вершины треугольника имеют один и тот же цвет и никаких других условий проверять не надо.
2. Используется Z-буфер, вершины треугольника имеют разный цвет и никаких других условий проверять не надо.

Использование такого набора специальных встраиваемых функций позволило дополнительно ускорить растеризацию примерно в 1.7 раза.

Реализация предложенных походов к алгоритмам растеризации, учитывающих специфику авиационных приложений и используемого процессора, позволила ускорить визуализацию PFD приложения на рис. 1 до ~12 кадров в секунду, а визуализацию рельефа на рис. 2 до скорости ~8 кадров в секунду. При этом все вычисления производились в одном потоке.

3. Использование многопотоковых вычислений

Использование многопотоковых вычислений для увеличения производительности библиотеки OpenGL является естественным развитием, поскольку большинство используемых процессоров являются многоядерными. Естественным подходом является разбиение экрана на прямоугольные области (тайлы) и обработка их различными потоками. Такой подход является классическим для использования многопотоковых вычислений при синтезе изображений с помощью трассировки лучей. Однако наши эксперименты показали неэффективность этого подхода. Причин здесь две. Первая – это дополнительные расходы на синхронизацию потоков. Вторая причина, не менее существенная для 2D авиационных приложений, это расходы на отработку отсечения треугольников границами прямоугольников. В случае отсутствия разбиения экрана на прямоугольники такой проблемы обычно не возникает, поскольку все треугольники, как правило, целиком попадают в область экрана.

В силу этих причин для распараллеливания растеризации мы использовали отдельные вычислительные потоки для построения изображения для разных кадров. В этом случае решаются обе вышеупомянутые проблемы падения производительности. Близкий подход использовался в работе [12]. Количество вычислительных потоков, используемых для растеризации, как правило, выбирается равным количеству ядер процессора. Дополнительно мы используем отдельный поток для копирования изображений, построенных вычислительными потоками, в буфер экрана (память видеокарты), поскольку затраты на эту процедуру являются критическими с точки зрения скорости визуализации. Поэтому ее следует выполнять параллельно с другими вычислениями, необходимыми для растеризации. Поскольку копирование изображения выполняется для кадра, для которого растеризация уже закончена, то никаких проблем с синхронизацией потоков не возникает. Синхронизация главного потока (main thread), вычислительных потоков непосредственно реализующих растеризацию и потока копирования изображений осуществляется с помощью событий. Используются следующие события:

1. *N* событий **s_ev[i]** для синхронизации начала работы i-го вычислительного потока;
2. *N* событий **e_ev[i]** сигнализирующих о завершении растеризации кадра i –ым вычислительным потоком;
3. *N* событий **c_ev[i]** сигнализирующих о готовности изображения, построенного i –ым вычислительным потоком, для копирования на экран.

Как уже было упомянуто выше, *N* обычно выбирается равным количеству ядер процессора. На этапе инициализации создаются N событий **s_ev, e_ev** и **c_ev.** Затем события **s_ev** и **c_ev** сбрасываются, а **e_ev** взводятся. Создаются *N* вычислительных потоков для растеризации *N* контекстов **cont[i]** для хранения информации о вызовах OpenGL и изображений, получаемых в результате растеризации вычислительными потоками. Создается также отдельный поток для копирования изображений, построенных вычислительными потоками, в буфер экрана. Индекс обрабатываемого кадра в главном потоке *m_idx* и индекс обрабатываемого кадра в потоке копирования изображений *c_idx* первоначально устанавливаются в 0. Наращивание индексов в процессе визуализации будет происходить по модулю *N*.

Алгоритм работы главного потока приведен на рис. 3:

1. Wait(**e_ev**[*m_idx*])
2. Обработка OpenGL вызовов => **cont**[*m_idx*]
3. Set(**s_ev**[*m_idx*])
4. *m_idx* = (*m_idx* ++) % *N* и возврат к п.1

Рис. 3. Алгоритм работы главного потока (main thread).

Поскольку все события **e_ev** взведены при инициализации *m_idx = 0*, то главный поток начинает сразу обрабатывать вызовы OpenGL для первого кадра и записывать результаты в контекст **cont**[*m_idx*]. По окончанию вызовов для данного кадра он взводит событие **s_ev**[*m_idx*] для запуска соответствующего вычислительного потока, переходит к следующему кадру по модулю *N* и снова ждет взведения события **e_ev**[*m_idx*].

Алгоритм работы вычислительного потока *idx* (*idx* = 0, 1, … *N*), производящего растеризацию, приведен на рис. 4:

1. Wait(**s_ev**[*idx*])
2. Обработка **cont**[*idx*] => изображение
3. Set(**c_ev**[*idx*])
4. Reset(**s_ev**[*idx*]) и возврат к п.1

Рис. 4. Алгоритм работы вычислительного потока *idx*.

Работа вычислительного потока *idx*, производящего растеризацию, начинается после взведения события s_**ev**[*idx*] главным потоком после заполнения контекста **cont**[*idx*]. Поток обрабатывает данные в контексте и строит из них путем растеризации треугольников, линий и точек изображение в памяти контекста. После этого он взводит событие **c_ev**[*idx*], сбрасывает событие **s_ev**[*idx*] и возвращается к п.1 ждать взведения события **s_ev**[*idx*].

Алгоритм работы потока, производящего копирование изображений из памяти контекста в память видеокарты, приведен на рис. 5:

1. Wait(**c_ev**[*c_idx*])
2. Копирование изображения **cont**[*c_idx*] => экран
3. Reset(**c_ev**[*c_idx*])
4. Set(**e_ev**[*c_idx*])
5. *c_idx* = (*c_idx* ++) % *N* и возврат к п.1

Рис. 5. Алгоритм работы потока копирования изображений

Работа потока копирования изображения начинается после взведения события **c_ev[***c_idx*] соответствующим вычислительным потоком. При инициализации индекс *c_idx* установлен в 0. После копирования изображения из контекста **cont**[*c_idx*] на экран (в память видеокарты) поток сбрасывает событие **c_ev**[*c_idx*], взводит событие **e_ev**[*c_idx*], наращивает индекс кадра *c_idx* по модулю *N* и снова ждет взведения события **c_ev**[c_*idx*].

Зависимость скорости растеризации (число кадров в секунду) от числа используемых потоков для описанного алгоритма для 4х ядерного процессора PowerPC приведена на рис. 6. Красный график показывает зависимость для 3D приложения (визуализация рельефа), синий график – для PFD приложения.

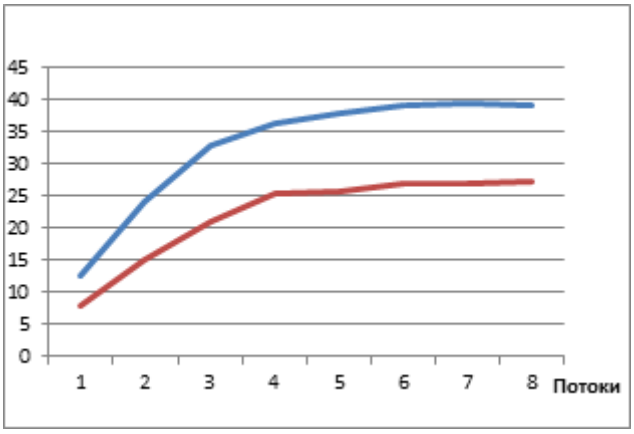

Рис. 6. Зависимость скорости растеризации (кадры в секунду) от числа используемых потоков.

Использование 4х потоков обеспечивает ускорение растеризации примерно в 2.9 - 3.2 раза.

Отдельно проведенное тестирование показало, что только копирование изображения из памяти на экран (в память видеокарты) занимает ~0.0165 сек для одного кадра, а работа приложения вместе с записью вызовов OpenGL в контекст и копированием посылаемых данных занимает ~0.006849 сек. Суммарное время ~0.023349 сек. Таким образом, получаем оценку сверху для производительности данного приложения ~42.83 кадра в секунду.

После оптимизации функций растеризации в п.2 и использования многопотоковых вычислений для них критическим, с точки зрения производительности, оказалась процедура копирования построенного изображения из памяти процессора в память видеокарты. По этой причине мы вынесли эту процедуру в отдельный поток с более высоким приоритетом. И если при реализации этой процедуры в главном потоке была достигнута скорость ~28 кадров в секунду, то при выделении для нее отдельного потока удалось достичь скорости ~36.5 кадров в секунду.

4. Результаты

Проведенные исследования показали возможность реализации программной OpenGL библиотеки на типичном авиационном компьютере на базе процессора PowerPC e500mc, 4 ядра, 1 ГГц, с производительностью, удовлетворяющей требованиям авиационных приложений. При этом библиотека полностью написана на языке C, не использует никаких специфических команд процессора PowerPC и переносима на любой компьютер, на котором поддерживается компилятор языка C. Процесс разработки библиотеки строго контролируется в соответствии с требованиями DO-178C, что в дальнейшем предоставляет возможность сертификации данной реализации OpenGL.

5. Благодарности

Работа поддержана грантами РФФИ 15-01-01147 и 16-01-00552.

6. Литература

[1] Федосов Е.А., Ковернинский И.В., Кан А.В., Солоделов Ю.А., Применение операционных систем реального времени в интегрированной модульной авионике. OSDAY 2015, http://osday.ru/solodelov.html.

[2] Safety Critical Working Group, https://www.khronos.org/openglsc/.

[3] ARINC Specification 661-6: http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=2698.

[4] DO-178C Software Considerations in Airborne Systems and Equipment Certification, http://www.rtca.org/store_product.asp?prodid=803.

[5] ArgusCore SC™ OpenGL SC 1.0.1 / SC 2.0 graphics drivers for Safety Critical Systems, http://www.coreavi.com/sites/default/files/coreavi_product_brief_-_arguscore_sc1_sc2_rev_a_0.pdf.

[6] P. M. a. J. Dudra, ADVANCED 2D RASTERIZATION ON MODERN CPUS // Applied Information Science, Engineering and Technology: Selected Topics from the Field of Production Information, т. 7, № 5, 2014.

[7] Маллачиев К.М., Пакулин Н.В., Хорошилов А.В., Устройство и архитектура операционной системы реального времени // Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 181-192 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-12.

[8] G. Wihlidal, Optimizing the Graphics Pipeline with Compute // GDC, 2016.

[9] T. K. Samuli Laine, High-Performance Software Rasterization on GPUs // High-Performance Graphics, 2011.

[10] The Mesa 3D Graphics Library, http://www.mesa3d.org.

[11] Модуль универсального процессора данных МУПД/Р3041-VPX 3U, http://www.nkbvs.ru/products/elektronnie-modyli/vpx-3u/moduli-universalnogo-protsessora-dannix-mypd-p3041/.

[12] А.Н. Милов, Особенности построения архитектуры масштабируемой графической системы стандарта OpenGL на основе ЦПОС // Труды 17-ой международной конференции по компьютерной графике и зрению, Россия, Московский Государственный Университет, июнь 23-27, 2007, с. 281-284.