Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»
Физтех-школа прикладной математики и информатики
Центр обучения проектированию и разработке игр

Направление подготовки: 09.04.01 Информатика и вычислительная техника

Направленность (профиль) подготовки: Анализ данных и разработка информационных систем

Оптимизация потребления видеопамяти при помощи вычислительного графа в приложениях реального времени

(магистерская диссертация)

Студент:

Санду Роман Александрович

_____

(подпись студента)

Научный руководитель:

Щербаков Александр Станиславович

_____

(подпись научного руководителя)

Москва 2023

Аннотация

Данная работа посвящена подходу к построению архитектуры приложений реального времени, называемому «графом отрисовки кадра», в частности вопросу оптимизации потребления видеопамяти транзиентными ресурсами используя преимущества этой архитектуры. Структура программы представляется как вычислительный граф, что позволяет автоматизировать различные аспекты разработки современных приложений реального времени, сводя их к алгоритмическим задачам. В рамках работы оптимизация потребления памяти сводится к обобщению широко известной задачи динамической аллокации памяти, проводится эффективный алгоритм решения этой задачи, а также освещаются различные инженерные аспекты имплементации системы графов отрисовки кадра.

Содержание

1. Введение

Около 10 лет назад история развития компьютерной графики реального времени потерпела кардинальный поворот с выходом графических API нового поколения, DirectX12, Vulkan и Metal, в 2014, 2016 и 2014 годах соответственно. Их предшественники, OpenGL и DirectX ранних версий, основывались на идее так называемого «толстого» драйвера. Дизайн этих API старался максимально скрыть принципы работы видеокарт, предоставляя пользователям простой, но достаточно ограниченный инструмент для разработки графических приложений реального времени. По мере развития индустрии компьютерной графики, разработчики приложений всё чаще сталкивались с ограничениями старых API, а простота дизайна всё больше жертвовалась в пользу поддержки новых возможностей графических ускорителей.

Обусловлен этот процесс возрастающими требованиями к приложениям со стороны клиентов. В индустриях игр, кино, виртуальных тренировочных симуляторов, а также промышленной визуализации, очень важны качество и реализм картинки. За годы академических исследований компьютерной графики было предложено большое количество техник и алгоритмов визуализации самых различных сцен, а развитие потребительской и промышленной техники привело к распространению широкого спектра устройств для запуска приложений. Так, не редка ситуация в которой одно и то же приложение должно масштабироваться от профессиональных персональных компьютеров с мощнейшими процессорами, видеокартами и мониторами в разрешении 8K, и до потребительских HMD-устройств, и даже телефонов и портативных консолей. От разработчиков приложений ожидается масштабируемость на этот широкий спектр устройств, с учётом всей их специфики, при этом добиваясь максимального возможного реализма картинки для конкретного устройства, не забывая так же о бизнес-требованиях к производительности.

Главной целью дизайна графических API нового поколения было позволить разработчикам оправдать эти ожидания, в следствии чего произошёл отказ от «толстых» драйверов в пользу раскрытия всё большего числа деталей работы графических ускорителей. Доступ к низкоуровневым механизмам графических ускорителей позволил приложениям лучше адаптироваться под конкретные устройства, рациональнее использовать доступные ресурсы, и как следствие, достигать большего реализма и производительности. Однако с большой властью приходит большая ответственность: сложность новых API требует от разработчика сильно

более структурированного подхода к разработки приложений, нежели старые, разработки собственных абстракций, собственных алгоритмов управления различными ресурсами. В следующем подразделе более детально освещены различные аспекты перехода к новым API, а также проблемы, возникающие при разработке приложений с их использованием.

1.1. Аспекты разработки графических приложений реального времени в старых и новых API

1.1.1. Управление памятью транзиентных ресурсов GPU

В процессе вычисления картинки одного кадра любое нетривиальное приложение использует *транзиентные ресурсы* – промежуточные хранилища данных, содержимое которых не требуется после окончания вычисления кадра, либо, требуется лишь в процессе вычисления следующего кадра. Как правило, подобные ресурсы являются картинками с разрешением кратным разрешению монитора пользователя. Из этого следует, что при переходе от 1080p мониторов к 4K мониторам потребление памяти транзиентными ресурсами возрастает в 4 раза, что обуславливает нужду в эффективном её переиспользовании. Старые графические API полностью скрывали управление памятью GPU от пользователя, предоставляя лишь функции создания и удаления конкретных ресурсов. За годы существования этой абстракции образовалось 3 основных подхода к эффективному управлению памятью транзиентных ресурсов.

Самый простым подходом является выделение и освобождение транзиентных ресурсов по ходу их нужды при помощи соответствующих вызовов графического API. Этот подход фактически идентичен выделению памяти в различных языках программирования: драйвер операционной системы содержит аллокатор, на который пользователь перекладывает обязанность управления памятью и другими ресурсами GPU, аналогично аллокациям на куче в языке C. Системный аллокатор переиспользует освободившуюся память, тем самым достигая низкого её потребления. Однако, такой подход не масштабируется на более сложные приложения. Во-первых известны нижние оценки на качество работы аллокаторов реального времени **1.ссылка на оценку онлайн DSA**, на практике выражающиеся как фрагментация кучи. Во-вторых, как правило и создание и освобождение ресурсов является весьма дорогой операцией в следствии деталей реализации драйверов. **2.мб объяснить почему?**

Альтернативным подходом служит отказ от переиспользования памяти. Все транзиентные ресурсы создаются заранее и не удаляются в ходе работы приложения. Очевидно, что с повышением сложности приложения такой подход перестаёт быть применимым, что иногда влечёт к попыткам в ручную переиспользовать некоторые выделенные объекты. Это, в свою очередь, приводит к чрезвычайно сложному для понимания коду, усложняя работу над самим приложением.

Наконец, наиболее практичным подходом является *пулирование* ресурсов. Вся программа работает с объектом называемым *пулом*, отвечающим за выделение и освобождение ресурсов. Пул использует аллокатор драйвера для выделения новых ресурсов, но вместо освобождения ресурсов в драйвер хранит список неиспользуемых ресурсов конкретного типа (в понятие тип как правило входит разрешение для текстур и размер для буфферов соответственно, а также все флаги свойств ресурса). Последующие запросы на выделение ресурсов обслуживаются в первую очередь из списка неиспользуемых, и только исчерпав его выделяются новые посредством драйвера операционной системы. Данный подход был оптимален до появления современных низкоуровневых графических API, однако в настоящее время хорошо заметен его главный недостаток: память не переиспользуется между ресурсами разных типов.

В современных же графических API предоставляется прямой, неограниченный доступ к памяти GPU. Приложение может выделять *кучи*, последовательности страниц виртуальной видеопамяти, и затем создавать ресурсы на конкретных адресах в рамках конкретной кучи. Стоит отметить, что пересечение используемых разными ресурсами регионов кучи не запрещается, хоть поведение при одновременном использование таких ресурсов не определено. Фактически, это нововведение перекладывает ответственность по написанию аллокатора ресурсов с разработчиков драйвера на разработчиков приложения, что можно сравнить с разработкой на языке C используя лишь системный вызов `malloc`. С одной стороны, это сильно усложняет разработку простых приложений. В следствии этого компания AMD открыла исходный код аллокатора из своих драйверов **3.ссылка**, к использованию которого нередко прибегают даже в промышленных приложениях, что, конечно же, возвращает статус-кво старых графических API. С другой стороны, это даёт возможность разработчикам более эффективно распоряжаться видеопамятью в различных подсистемах приложения, в частности, позволяя

построить в некотором смысле оптимальное расписание аллокации транзиентных ресурсов.

1.1.2. Отправка команд GPU

Работа с любыми внешними по отношению к центральному процессору устройствами по своей природе асинхронна. Передача данных по проводам занимает время, как и их обработка на внешнем устройстве. Заставлять ядра центрального процессора простаивать в ожидании отклика от внешнего устройства – непозволительная растрата ресурсов. Не исключение и графические ускорители. Низкоуровневым инструментом для общения GPU и операционной системы служат *списки команд*, состоящии из команд отрисовки, запуска вычислений, синхронизации, копирования данных, и прочих. В случае если центральному процессору необходимо дождаться результата каких либо вычислений на GPU, ожидание необходимо делать вручную, используя аппаратные сигналы о прогрессе от видеоускорителя.

В старых API асинхронная природа вычислений на GPU скрывалась за абстракцией *мгновенного режима* (от англ. immediate mode). Драйвер создавал видимость мгновенного выполнения всех команд GPU, представлявших собой функции. Примитивы синхронизации внутри GPU, а также между GPU и CPU вставлялись автоматически, что не редко приводило к непредсказуемой производительности кода. Более того, производительность крупных приложений могла упираться в скорость записи командных буферов внутри драйвера. Естественным способом решения этой проблемы была бы параллельная их запись, но машина состояний внутри драйверов старых графических API была фундаментально однопоточной структурой. Далее, в какой-то момент времени графически ускорители начали поддерживать параллельную обработку нескольких очередей команд. Эта возможность может давать прирост производительности при слабой загрузке вычислительных модулей GPU исполняемыми командами. Дизайн старых API не был рассчитан на поддержку таких возможностей аппаратуры, что сильно усложняло работу с ними.

С приходом новых графических API фактически все проблемы с отправкой команд были решены. Новые API предоставляют пользователю прямой доступ к спискам команд, примитивам синхронизации и очередям исполнения команд. Однако, как и в случае с прямым доступом к памяти, от разработчиков требуется некоторый уровень дисциплины при работе с предоставляемыми абстракциями. **4.Как-то оборвано, возможно стоит плавнее перейти к записи команд буферов**

через фреймграф. Или вообще не говорить про это всё?

1.1.3. Управление кешами GPU и состоянием ресурсов

Характерным отличием графических ускорителей от центральных процессоров является отсутствие гарантий когерентности кешей, и в следствии чего необходимость в ручную делать их инвалидацию и сброс. Однако старые графические API скрывали эту особенность аппаратуры, автоматически отслеживая состояние ресурса и вставляя соответствующие команды синхронизации в список команд автоматически. Недостаток такого подхода заключается в отсутствии невозможности предсказывать последующие команды пользователя на уровне драйвера, что не редко приводит к исполнению команд синхронизации в неоптимальный момент времени. Усугубляет ситуацию тот факт, что некоторые GPU используют различные оптимизации формата хранения ресурсов при их использовании конкретным образом, и переход между разными оптимизированными состояниями приводит к простою вычислительных ресурсов GPU.

В новых же графических API ответственность за отслеживание состояния ресурсов и кешей переложена на пользователя посредством абстракции *барьеров*. Барьеры служат главным примитивом синхронизации в рамках GPU, управления кешами и состояниями ресурсов. Однако расстановка барьеров в корректных и оптимальных местах порой оказывается далеко не тривиальной задачей, требующей от пользователя глобального понимания работы всей системы. Это делает модуляризацию приложения невозможным без введения специальных механизмов для расстановки барьеров.

1.2. Кадровый граф

Сложно отследить появление понятия вычислительного графа, однако первым приложением этой техники в контексте графических приложений реального времени считается игровой движок Frostbite компании EA, о чём было объявлено в 2017 году на конференции Game Developers Conference [1]. С тех пор большая часть коммерческих движков перешла на архитектуру основанную на кадровом графе, о чём подробнее в следующем разделе. Причина этого перехода – наличие кадрового графа в виде данных позволяет решить все вышеописанные проблемы, появившиеся с приходом новых графических API, при этом сделать это оптимальнее чем позволяли встроенные в драйвер механизмы старых API.

Уточним используемую в дальнейшем терминологию. *Кадровым графом* назовём конкретный набор вершин, рёбер и других данных, определяемый настройками и спецификой приложения, и задающий глобальную структуру процесса вычисления одного кадра. *Рантаймом* кадровых графов будем называть программное решение, принимающее на вход конкретный кадровый граф и позволяющее его *компилировать*, запускать и, возможно, редактировать. Процесс компиляции кадрового графа – некоторый набор действий и вычислений, которые необходимо совершить перед запуском кадрового графа. Конкретный набор данных, задающий кадровый граф, определяется дизайном конкретного рантайма, как и специфика процессов компиляции, запуска, а также дополнительный функционал, предоставляемый пользователям рантайма, а именно, разработчикам алгоритмов визуализации сцены. Пространство дизайна рантайма кадровых графов весьма широко, но независимо от конкретных решений, рантайм будет принимать на вход набор вершин, каждая из которых содержит *функцию запуска*, инициирующую некоторые вычисления на GPU, и список используемых функцией запуска ресурсов с дополнительной информацией о конкретном способе использования. Рёбра же графа как правило не задаются явно, а создаются автоматически, например, между вершиной создающей некоторый ресурс, и читающей этот ресурс вершиной. Абстрагируясь от конкретики, опишем как именно архитектура, основанная на кадровом графе, позволяет решить описанные выше проблемы.

Во-первых, имея в виде данных глобальную информацию об использовании ресурсов на протяжении всего кадра, вопрос управления памятью GPU сводится к неинтерактивной вариации хорошо известной в литературе задачи динамической аллокации памяти [2, с. 226]. В отсутствии глобальной информации управление памятью является интерактивной вариацией задачи о динамической аллокации памяти. Известно, что алгоритмы для интерактивной вариации этой задачи работают качественно хуже, чем для статической. В программной инженерии этот результат известен как проблема фрагментации кучи. Более того, как уже было упомянуто выше, создание ресурсов GPU – достаточно дорогая операция, а качественные алгоритмы решения интерактивной динамической аллокации памяти занимают значительное время. Эти факторы делают решение неинтерактивной вариации этой задачи один раз в момент компиляции кадрового графа привлекательной идеей.

Во-вторых, если рантайм требует от пользователей указывать конкретный способ исполь-

зования ресурсов внутри вершины, то в процессе запуска кадрового графа становится возможным автоматически расставлять барьеры. Если ресурс был создан и заполнен данными посредством рендеринга вершиной A, а затем после запуска некоторого количества других вершин был семплирован вершиной B, то рантайм обязан поставить барьер переводящий ресурс из состояния пригодного для рендеринга в состояние пригодное для семплирования. Однако из-за наличия нескольких промежуточных вершин, у рантайма есть выбор, в какой конкретно момент поставить барьер. Этот выбор может влиять на производительность из-за конвейеризации вычислений на современных графических ускорителях, а значит можно сформулировать задачу дискретной оптимизации расстановки барьеров с целью минимизации простоев вычислительных модулей GPU. В данной работе, однако, такая задача не рассматривается.

В-третьих, интересной особенностью современных графических API, не упомянутой ранее, является поддержка специфичной для мобильных устройств архитектуры графических ускорителей, называемой «Tile Based Deferred Renderer» (TBDR). Не вдаваясь в детяли принципов работы TBDR **5.может всё таки вдаться?**, старые графические API были абсолютно не рассчитаны на подобные графические ускорители, что приводило к непредсказуемому влиянию изменений в коде на производительность, а также делало некоторые техники визуализации вроде отложенного освещения неприменимыми на подобных устройствах. В новых же API был предоставлен почти полный контроль над механизмами работы TBDR, что с одной стороны решило эти проблемы, но с другой стороны сильно усложнило процесс разработки из-за введения понятия *рендер-пасса* [3, раздел 8]. Введения слоя абстракции над графическим API в виде кадрового графа позволяет одновременно и упростить интерфейс предоставляемый пользователю для работы с TBDR и рендер-пассами, и сохранить предсказуемость производительности, отчасти путём предоставления инструментов визуализации структуры графа.

6.Написать про:

- **7.архитектурную крутизну ресурсных зависимостей**

- **8.автоматический подбор хорошего для платформы порядка исполнения**

- **9.многопоточную запись команд**

- **10.автоматический асинк**

- **11.мультиплексирование и прочие ”циклы”**

- **12.историю ресурсов – наверное не надо, это же наш орижинал контент**

2. Обзор существующих работ

2.1. Комплексные решения

Frostbite

Первыми идею организации архитектуры рендеринга в приложениях реального времени через вычислительные графы предложили разработчики движка Frostbite в 2017 году[1]. *Кадровый граф* позволил им сделать ядро модуля рендеринга расширяемым, упростил работу с асинхронным вычислениями общего назначения на GPU, автоматизировал работу со специализированными видами оперативной видеопамяти на игровых консолях, а также сэкономил большое количество обычной видеопамяти. В силу проприетарности движка не известно, насколько широкий класс сценариев использования ресурсов она поддерживает. В качестве схемы аллокации ресурсов же был взят обычный онлайн-аллокатор, располагающий в заранее выделенном крупном участке памяти ресурсы по мере необходимости. Автоматическая расстановка барьеров на 2017 год не поддерживалась.

Halcyon

Далее, в 2019 году, компания EA представила[4] новый экспериментальный движок Halcyon, обобщающий идею кадрового графа до *графа рендеринга*. Как следует из названия, это обобщение позволяет организовывать в виде графа вычислений не только процесс рендеринга самого кадра, но и рендеринг различных вспомогательных изображений, например кубических карт для некоторых техник глобального освещения, изображений импосторов, или различных иконок в приложении. Более того, граф рендеринга может состоять из нескольких подграфов, запускающихся с разной частотой, например подграф, вычисляющий тени от солнца, может запускаться только при значительном изменении положения солнца на небе. В отличии от Frostbite, граф Halcyon поддерживает автоматическую расстановку барьеров. Также на выступлении отмечается, что изначально граф составлялся явной композицией вершин и подграфов, но в итоге разработчики пришли к дизайну с автоматической композицией вершин на основе глобально видимых имён ресурсов. Про алгоритм аллокации ресурсов и поддержку переживающих границу кадра ресурсов публично доступной информации нет.

Наконец, граф рендеринга Halcyon способен в автоматическом режиме масштабироваться на несколько аппаратных графических ускорителей, и даже на несколько компьютеров. Под-

держка такого функционала весьма сложна и говорит об экспериментальности этой разработки, так как на практике такая масштабируемость редко применима.

Unreal Engine

Начиная с версии 4.22 в Unreal Engine начал переходить на рендеринг через систему ”Render Dependency Graph”[5], представляющую собой граф рендеринга. Однако даже в 5й версии движка эта система использует жадную он-лайн стратегию аллокации ресурсов, хоть и поддерживает большое количество важного функционала: автоматизацию асинхронных вычислений на GPU, расстановку барьеров и параллелизацию исполнения графа.

Unity

Разработчики движка Unity, следуя общему направлению индустрии, в 2018 году перевели архитектуру рендеринга на подход вычислительных графов[6]. Однако среди прочих проприетарных движков про граф рендеринга Unity известно, пожалуй, меньше всего. Отметить стоит лишь наличие интеграции между нативным и скриптовым кодом рендеринга, позволяющей сократить время итерации при разработке приложений.

Anvil

Движок Anvil компании Ubisoft с переходом на DirectX12 тоже начал выделять подсистему зависимостей ресурсов, хоть и не называя её графом кадра[7][8]. Согласно выступлениям на GDC, эта система поддерживает многий функционал уже упомянутых: переиспользует память ресурсов, автоматически расставляет барьеры, автоматизирует асинхронные вычисления. Но как и в случае Unity, детали устройства отсутствуют в публичном доступе.

Render Pipeline Shaders

Выпущенная в открытый доступ[9] в декабре 2022 года библиотека Render Pipeline Shaders компании AMD в своём составе имеет комплексное решение для построение кадровых графов[10]. Эта библиотека полностью скрывает от пользователя управление транзиентными ресурсами посредством предметно-ориентированного языка, автоматически переиспользуя ресурсы, расставляя барьеры и клонируя объявленные единожды вершины графа. **13.ПРОЧИТАТЬ ИСХОДНИКИ И НАПИСАТЬ ЕШО**

Granite

Наконец, стоит упомянуть о существовании многих любительских проектов по написанию обобщённой библиотеки кадровых графов. Большинство из них находятся на стадии зарождения и не заслуживают подробного рассмотрения. Исключеним является проект Granite[11], в рамках которого разработан кадровый граф адаптированный для использования на мобильных устройствах посредством API Vulkan. Кадровый граф Granite автоматически расставляет примитивы синхронизации, группирует ноды в рендер-пассы [3, раздел 8], оптимизирует порядок исполнения вершин с точки зрения минимизации накладных расходов на синхронизацию, а также переиспользует память, хоть и при помощи жадного алгоритма аллокации.

2.2. Аллокация ресурсов

Задача поиска расписания аллокации ресурсов в графе кадра в своей простейшей формулировке является классической сильно NP-сложной[12] задачей *динамической аллокации памяти* (dynamic storage allocation, DSA[2, с. 226]). У этой задачи существует две интерпретации, он-лайн и офф-лайн. Первая подразумевает обработку разнесённых во времени запросов на аллокацию и деаллокацию ресурсов, иначе говоря, решения об адресах ресурсов в памяти необходимо принимать в порядке времён появления ресурсов. Этот частный случай часто встречается в операционных системах и рантаймах языков программирования. Вторая же интерпретация подразумевает наличие заранее известных времён жизни всех ресурсов. В рамках данной работы нас интересует именно офф-лайн интерпретация, поэтому, в отсутствие уточнения, под задачей о динамической аллокации памяти мы будем подразумевать именно её.

Одним из первых полиномиальных алгоритмов предложенных для решения задачи DSA является алгоритм First-Fit[13], работающий, как было вскоре доказано Кирстедом, с константной ошибкой не более чем в 80 раз[14]. Тремя годами позже Кирстед представил алгоритм с ошибкой не более чем в 6 раз[15]. Эти и другие ранние работы объединяет общий подход сведения DSA к частному случаю с единичным размером всех ресурсов, эквивалентному покраске интервального графа, и последующим применением он-лайн алгоритма покраски. Через несколько лет Йордан Гергов, отказавшись от сведения к интервальным графам, смог понизить верхнюю оценку минимальной возможной ошибки до 5[16], а в последствии и до 3[17]. Наконец, наилучший на данный момент результат был получен исследователями

из AT&T Labs совместно с коллегой из Ecole Polytechnique[18]: полиномиальный алгоритм, для любого заранее выбранного ε дающий $(2+\varepsilon)$-приблизительное решение DSA. Более того, для некоторых частных случаев авторы предоставляют приближённую схему полиномиального времени (то есть $(1+\varepsilon)$-приближение). Из них в рамках графа кадра особо интересна схема для случая ресурсов, размер которых ограничен сверху константой h_{max}. Однако практичность представленных алгоритмов в рамках приложений реального времени является открытым вопросом в силу их высокой сложности (TODO: оценить асимптотику по мастер-теореме).

Похожая задача возникает в области оперирования морских контейнерных терминалов. С ростом сложности и нагруженности глобальных транспортных цепочек, прикладные задачи оперирования верфей стали слишком сложны для интуитивного их решения. В связи с этим за последние несколько десятилетий было сформулировано и в той или иной степени решено множество вариаций *задачи об аллокации верфи*, покрывающих широкий спектр прикладных задач. Так как расписания прибытия кораблей обычно известно портам заранее, офф-лайн задача динамической аллокации памяти является частным случаем одной из формулировок этой задачи, а именно вариации классифицируемой в обзорной статье Бирвирта и Мизла[19] как $cont|dyn|fix|max(res)$. Именно из-за этого задача об аллокации верфи представляет интерес в рамках данной работы.

Одним из первых интересующую нас формулировку задачи об аллокации верфи рассмотрел в своей статье Эндрю Лим[20]. Ресурсы, имеющие фиксированные и известные размер и времена аллокации и деаллокации, могут быть рассмотрены как корабли с соответствующей длинной, временем прибытия и временем отплытия, а тип используемой видеопамяти как секция верфи. Задача нахождения минимальной длинны всех секций верфи и точек прибытия всех кораблей аналогична нахождению минимального необходимого объёма памяти и локаций всех ресурсов в этой памяти. Однако, в отличии от рассматриваемой Лимом задачи, ресурсы не накладывают требований на отступ между друг другом и началом или концом верфи, зато требуют определённого выравнивания их начала в памяти. Впрочем, последние условие достаточно легко сводится к первому.

Однако в данной работе рассматривается более общая формулировка задачи об аллокации

ресурсов, насколько известно автору, не рассматривавшаяся ранее в литературе.

Список литературы

1. *O'Donnell Y.* FrameGraph: Extensible Rendering Architecture in Frostbite. — 2017. — URL: https://www.gdcvault.com/play/1024612 ; Game Developers Conference.
2. *Garey M. R.*, *Johnson D. S.* Computers and Intractability; A Guide to the Theory of NP-Completeness. — USA : W. H. Freeman & Co., 1990. — ISBN 0716710455.
3. *Inc. T. K. G.* Vulkan API Specification. — URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html.
4. *Wihlidal G.* Halcyon: Rapid innovation using modern graphics. — 2019. — URL: https://www.youtube.com/watch?v=da_6dsWz8yg ; Reboot Develop.
5. *Games E.* Unreal Engine rebder dependency graph. — URL: https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/.
6. *Tatarchuk N.*, *Aaltonen S.*, *Cooper T.* Unity Rendering Architecture. — 2021. — URL: https://www.youtube.com/watch?v=6LzcXPIWUbc ; SIGGRAPH 2021 REAC.
7. *Gruen H.* DirectX™ 12 Case Studies. — 2017. — URL: https://www.gdcvault.com/play/1024343 ; Game Developers Conference.
8. *Rodrigues T.* Moving to DirectX 12: Lessons Learned. — 2017. — URL: https://www.gdcvault.com/play/1024656 ; Game Developers Conference.
9. *Advanced Micro Devices I.* Исходный код Render Pipeline Shaders. — URL: https://github.com/GPUOpen-LibrariesAndSDKs/RenderPipelineShaders.
10. *Advanced Micro Devices I.* Анонс публикации Render Pipeline Shaders. — URL: https://gpuopen.com/learn/rps_1_0.
11. *Arntzen H.-K.* Render graphs and Vulkan — a deep dive. — 2017. — URL: https://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/.
12. *Stockmeyer I. J.* — 1976. — личная переписка.
13. *Chrobak M.*, *Ślusarek M.* On some packing problem related to dynamic storage allocation // RAIRO - Theoretical Informatics and Applications. — 1988. — Vol. 22, no. 4. — P. 487–499. — ISSN 0988-3754, 1290-385X. — DOI: 10.1051/ita/1988220404871. — URL: http://www.rairo-ita.org/10.1051/ita/1988220404871 (visited on 10/23/2022).
14. *Kierstead H. A.* The Linearity of First-Fit Coloring of Interval Graphs // SIAM Journal on Discrete Mathematics. — 1988. — Nov. — Vol. 1, no. 4. — P. 526–530. — ISSN 0895-4801,

1095-7146. — DOI: 10.1137/0401048. — URL: http://epubs.siam.org/doi/10.1137/0401048 (visited on 10/23/2022).

15. *Kierstead H. A.* A polynomial time approximation algorithm for dynamic storage allocation // Discrete Mathematics. — 1991. — Т. 88, № 2. — С. 231—237. — Publisher: Elsevier.
16. *Gergov J.* Approximation algorithms for dynamic storage allocation // European Symposium on Algorithms. — Springer, 1996. — С. 52—61.
17. *Gergov J.* Algorithms for compile-time memory optimization // Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. — 1999. — С. 907—908.
18. OPT versus LOAD in dynamic storage allocation / A. L. Buchsbaum [и др.] // Proceedings of the thirty-fifth annual ACM symposium on Theory of computing. — 2003. — С. 556—564.
19. *Bierwirth C.*, *Meisel F.* A survey of berth allocation and quay crane scheduling problems in container terminals // European Journal of Operational Research. — 2010. — Т. 202, № 3. — С. 615—627. — ISSN 0377-2217. — DOI: https://doi.org/10.1016/j.ejor.2009.05.031. — URL: https://www.sciencedirect.com/science/article/pii/S0377221709003579.
20. *Lim A.* The berth planning problem // Operations Research Letters. — 1998. — Т. 22, № 2. — С. 105—110. — ISSN 0167-6377. — DOI: https://doi.org/10.1016/S0167-6377(98)00010-8. — URL: https://www.sciencedirect.com/science/article/pii/S0167637798000108.