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

С самого появления области интерактивных приложений реального времени с трёхмерной графикой, главной задачей является максимизация качества картинки без потери производительности. Так, основным бизнес-требованием от индустрий кино, игр, виртуальных симуляторов, промышленной визуализации, является реализм итогового изображения. С целью удовлетворения этих требований было проведено множество академических исследований различных методов визуализации, разработаны новые алгоритмы и техники отрисовки сцен. Параллельно с этим не стояла на месте и область потребительской и промышленной техники. В настоящий день не редка ситуация, когда одно и то же приложение должно запускаться на целом ряде различных устройств: персональных компьютерах абсолютно разной комплектации, стационарных и портативных игровых консолях, мобильных устройствах различных производителей и даже HMD-устройствах виртуальной и дополненной реальности[1].

Вопрос потребления видеопамяти такими приложениями является камнем преткновения для многих устройств. Достижение качественной картинки требует использования продвинутых методов визуализации, продвинутые методы визуализации требуют дополнительной видеопамяти, а устройств с неограниченной памятью человечеством на данный момент изобретено не было. Наиболее остро вопрос расхода видеопамяти стоит на портативных устройствах: смартфонах, консолях, HMD-устройствах. Среди последних, например, устройства серии «Quest» содержат 4 и 6 гигабайт гибридной памяти, используемой и GPU, и CPU. С суммарным разрешением 2x1832x1920, всего одно полноэкранное изображение на «Quest 2» занимает около 113 мегабайт. Предположив равное использование памяти GPU и CPU, а также резервацию нескольких гигабайт под данные сцены, приложению вряд ли удастся создать

[1] Устройства, закрепляемые на голове пользователя, от англ. head mounted device.

больше 10 полноэкранных изображений для использования алгоритмами визуализации, 4–5 из которых, скорее всего, сразу же будут использованы G-буфером распространённой техники отложенного освещения [**10.1145/54852.378468**]. С другой стороны, наименее проблематичной категорией устройств считаются персональные компьютеры. Согласно обзору ПК пользователей «Steam» [1], медианным количеством видеопамяти являются 8 гигабайт, а медианным разрешением 1920x1080. Конечно же, такая ситуация более благоприятна для сложных методов визуализации сцен. Однако необходимо учитывать, что 1–2 гигабайта видеопамяти зачастую расходуется прочими приложениями, запущенными пользователем и операционной системой, а ожидания пользователей о качестве картинки и детализации сцены много выше чем в случае с HMD-устройствами. Более того, согласно [1], на данный момент более 20% пользователей «Steam» пользуются мониторами в 4K разрешении, требующими в 4 раза большей детализации от приложения. Для оправдания всех ожиданий и требований, от разработчиков приложений требуется эффективное использование доступной видеопамяти.

Тем не менее, до относительно недавнего времени, эффективное управление видеопамятью было попросту невозможно в силу технических ограничений, рассмотрению которых посвящён следующий раздел.

2. Введение в предметную область

Около 10 лет назад история развития компьютерной графики реального времени потерпела кардинальный поворот с выходом графических API нового поколения, DirectX12, Vulkan и Metal. 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 подобной архитектуры, а простота дизайна всё больше жертвовалась в пользу поддержки новых возможностей различных типов графических ускорителей. Так, например, печально известна неоправданная сложность разработки трёхмерных приложений для мобильных платформ с использованием «OpenGL for Embedded Systems».

Главной целью дизайна графических API нового поколения было избавление от накладных расходов сложных абстракций, вследствие чего произошёл отказ от «толстых» драйверов в пользу раскрытия всё большего числа внутренних деталей работы GPU. Доступ к низкоуровневым механизмам графических ускорителей позволил приложениям лучше адаптироваться под конкретные устройства, рациональнее использовать доступные ресурсы, и, как следствие, достигать большего качества и производительности. Однако сложность новых API требует от разработчика сильно более структурированного подхода к разработке приложений, дизайна собственных абстракций, собственных алгоритмов и систем 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.

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

2.1.1. Управление памятью транзиентных ресурсов

Главный интерес в рамках данной работы составляют кардинальные изменения в работе с видеопамятью. 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.

Самым 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 интерактивных аллокаторов (см. [2]), на практике выра-

жающиеся как фрагментация кучи. Во-вторых, как правило, и создание, и освобождение ресурсов являются весьма дорогими операциями вследствие деталей реализации драйверов. 1.мб объяснить почему?

В некотором смысле противоположным 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 приводит к попыткам вручную переиспользовать некоторые выделенные объекты. Это, в свою очередь, приводит к чрезвычайно сложному для понимания коду, усложняя и замедляя работу над самим приложением.

Наконец, наиболее практичным подходом является техника *пулирования* ресурсов. Вся программа работает с объектом называемым *пулом*, отвечающим за выделение и освобождение ресурсов. Пул использует механизмы 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 буферов 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 ресурсы 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 запреща-

ется, хоть поведение при одновременном использовании 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 системные вызовы POSIX `mmap` и `munmap` вместо аллокатора библиотеки glibc. С одной стороны, это сильно усложняет разработку простых приложений. Вследствие этого компания AMD открыла исходный код аллокатора из своих драйверов [3], 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, чему в первую очередь и посвящена данная работа.

2.1.2. Отправка команд GPU

Работа с любыми внешними по отношению к центральному процессору компонентами компьютера 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*, состоящие 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 в рамках 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 графические 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 этим функционалом.

С приходом новых графических API фактически все проблемы с отправкой команд были решены, а точнее ответственность по их решению была переложена на разработчиков приложений. 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, работа напрямую с предоставляемыми графическим API абстракциями не является практичной, необходима разработка собственных абстракций для удобной записи команд.

2.1.3. Управление кешами GPU и состоянием ресурсов

Характерным отличием графических ускорителей от центральных процессоров является отсутствие гарантий когерентности кешей, и вследствие чего необходимость в ручную делать их инвалидацию и сброс. Однако ста-

рые графические API скрывали эту особенность аппаратуры, автоматически отслеживая состояние ресурса и вставляя соответствующие команды синхронизации в список команд. Недостаток такого подхода заключается в отсутствие 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.

2.2. Кадровые графы

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 [4]. С тех пор большая часть коммерческих движков перешла на архитектуру, основанную на кадровом графе, о чём подробнее в следующем разделе. Причина этого перехода следующая: 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.

2.2.1. Управление памятью транзиентных ресурсов в кадровых графах

Во-первых, имея в виде данных глобальную информацию об использовании ресурсов на протяжении всего кадра, вопрос управления памятью GPU

сводиться к неинтерактивной вариации хорошо известной в литературе задачи динамической аллокации памяти [5, с. 226]. В отсутствие же глобальной информации управление памятью является интерактивной вариацией этой задачи. Как уже было упомянуто ранее, известно, что алгоритмы для интерактивной вариации этой задачи работают качественно хуже, чем для статической (см. [2]), и более того, 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 единожды в момент компиляции кадрового графа привлекательной идеей, и наоборот, скорее всего любое решение оптимально управляющее памятью транзиентных ресурсов в том или ином виде будет сводится к кадровому графу.

2.2.2. Управление кешами и состоянием ресурсов в кадровых графах

Во-вторых, если рантайм требует от пользователей указывать конкретный способ использования 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. Впрочем, постановка и решение такой задачи выходит за рамки данной работы.

2.2.3. Кадровые графы и GPU архитектуры TBDR

В-третьих, важной особенностью современных графических API, не упомянутой ранее, является поддержка специфичной для портативных и мобильных устройств архитектуры графических ускорителей, называемой «Tile Based Deferred Renderer» (TBDR). Не вдаваясь в детали принципов работы TBDR, старые графические API были абсолютно не рассчитаны на подобные возможности апаратуры, 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* [6, раздел 8]. Введение 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.

2.2.4. Отправка команд GPU из кадрового графа

В-четвёртых, как уже было упомянуто выше, современные GPU часто предоставляют разработчикам несколько очередей для отправки команд, что позволяет лучше насытить вычислительные модули в ситуации когда выполняемые команды не требуют стопроцентного использования всех возможностей ускорителя. Рантайм кадровых графов может помочь автоматизировать этот процесс, заранее выбирая стратегию планирования имеющихся вершин на имеющиеся очереди команд. Далее, даже в рамках одной очереди команд не редко имеет смысл чередовать команды от независимых ветвей графа с целью лучше насытить вычислительные ресурсы GPU полезной работой. Более

того, само исполнение вершин кадрового графа для получения списков команд рантайм может производить параллельно для независимых ветвей графа, что может понизить время затрачиваемое на кадр на центральном процессоре. В данной работе, однако, более глубоко данный вопрос не освещается.

2.2.5. Архитектурные аспекты кадровых графов

Наконец, вероятно самым важным фактором в распространении кадровых графов послужили архитектурные преимущества этого подхода. Как было сказано ранее, рёбра графа обычно задаются пользователями рантайма неявно, посредством ресурсных зависимостей. Это позволяет добиться низкой связности различных подсистем приложения: модулям не требуется знать о прочей системе ничего, кроме названий ресурсов и публичного API рантайма кадровых графов. Более того, рантайм может автоматически обнаруживать некорректные конфигурации приложения, в которых тем или иным модулям не хватает необходимых ресурсных зависимостей, отключая соответствующие модули автоматически.

Конечно же, далеко не все имплементации рантаймов кадровых графов содержат решения всех перечисленных проблем, и не всегда в описанном выше виде. Некоторые же имплементации содержат уникальный для них функционал, решающий специфичные для конкретной области проблемы. Обзор существующих комплексных решений содержится в следующем разделе работы.

3. Цели и задачи

Цель данной работы — разработка схемы оптимизации потребления памяти транзиентными ресурсами, в том числе используемыми в темпоральных алгоритмах. Исходя из этого, был поставлен следующий ряд задач:

— обзор существующих решений,

— формулировка алгоритмической задачи управления видеопамятью,

— разработка качественного метода решения задачи управления видеопамятью,

— проведение анализа потребления памяти и производительности.

Как уже было сказано выше, оптимальное управление видеопамятью требует глобальных знаний о процессе вычисления кадра, что в свою очередь фактически эквивалентно использованию кадрового графа. В силу этого удовлетворительное с практической точки зрения решение последних трёх задач также требует решения следующих инженерных подзадач:

— разработка рантайма кадровых графов в рамках движка Dagor, обладающего достаточным функционалом для интеграции в существующие проекты;

— разработка эргономичного пользовательского API для рантайма;

— непосредственная интеграция разработанного решения в проект, основанный на движке Dagor, «Enlisted».

4. Обзор существующих работ

4.1. Рантаймы кадровых графов

Переходя к обзору различных имплементаций рантайма кадровых графов, стоит отметить, что далеко не все компании готовы рассказывать об используемых в их разработках технологиях. Список, приведённый ниже, был составлен, основываясь на открытых источниках информации, и не претендует на полноту.

Frostbite

Первыми идею организации архитектуры рендеринга в приложениях реального времени через вычислительные графы предложили разработчики движка Frostbite в 2017 году [4]. *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 неизвестно, насколько широкий класс сценариев использования ресурсов она поддерживает. В качестве схемы аллокации ресурсов же был взят обычный интерактивный 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 [7] 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 импостеров, 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 отличие 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. Поддержка 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”[8], представляющую собой граф рендеринга. Однако даже в пятой версии движка эта система использует жадную интерактивную 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 [9]. 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 [10; 11]. 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 [12] в декабре 2022 года библиотека Render Pipeline Shaders компании AMD в своём составе имеет комплексное решение для построения кадровых графов [13]. Эта библиотека полностью скрывает от пользователя управление транзиентными ресурсами посредством предметно-ориентированного языка «RPSL», автоматически переиспользуя ресурсы, расставляя барьеры и клонируя объявленные единожды вершины графа. Разработанное AMD решение также поддерживает стандартный функционал кадровых графов: экономия памяти посредством интерактивного алгоритма аллокации, расстановку барьеров и асинхронные вычисления. Наибольшим его преимуществом, вероятно, является исходный открытый код.

Недостатком этой разработки является отсутствие переиспользования памяти *темпоральных* ресурсов. В терминах данной работы, любые ресурсы, чья история была запрошена, исключаются из алгоритма аллокации и хранятся в отдельных аллокациях.

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. Исключением является проект Granite [14], в рамках которого разработан кадровый граф, адаптированный для использования на мобильных устройствах посредством API Vulkan. Кадровый граф Granite автоматически расставляет примитивы синхронизации, группирует вершины в рендер-пассы [6, 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.

4.2. Аллокация ресурсов

Задача поиска расписания аллокации ресурсов в графе кадра в своей простейшей формулировке является классической сильно NP-сложной [15] задачей *динамической аллокации памяти* (dynamic storage allocation, DSA [5, с. 226]). У этой задачи существует две интерпретации, интерактивная и неинтерактивная. 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 неинтерактивная 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 [16], работающий, как было вскоре доказано Кирстедом, с константной ошибкой не более чем в 80 раз [17]. Тремя годами позже Кирстед представил алгоритм с ошибкой не более чем в 6 раз[18]. 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 интерактивного алгоритма покраски. Через несколько лет Йордан Гергов, отказавшись от сведения к интервальным графам, смог понизить верхнюю оценку минимальной возможной ошибки до 5 [19], а впоследствии и до 3[20]. Наконец, наилучший на данный момент результат был получен исследователями из AT&T Labs совместно с коллегой из Ecole Polytechnique [21]: 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, а также асимптотического характера теоретических оценок ошибки.

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 множе-

ство вариаций *задачи об аллокации верфи*, покрывающих широкий спектр прикладных задач логистики. Так как расписания прибытия кораблей обычно известно портам заранее, неинтерактивная задача динамической аллокации памяти является частным случаем одной из формулировок этой задачи, а именно вариации классифицируемой в обзорной статье Бирвирта и Мизла [22] как $cont|dyn|fix|max(res)$. Именно из-за этого задача об аллокации верфи представляет интерес в рамках данной работы. **2.Может нафиг эти верфи?**

Одним из первых интересующую нас формулировку задачи об аллокации верфи рассмотрел в своей статье Эндрю Лим [23]. Ресурсы, имеющие фиксированные и известные размер и времена аллокации и деаллокации, могут быть рассмотрены как корабли с соответствующей длиной, временем прибытия и временем отплытия, а тип используемой видеопамяти как секция верфи. Задача нахождения минимальной длины всех секций верфи и точек прибытия всех кораблей аналогична нахождению минимального необходимого объёма памяти и локаций всех ресурсов в этой памяти. Однако в отличие 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, позволяющая также переиспользовать память ресурсов, используемых темпоральными алгоритмами, и, насколько известно автору, не рассматривавшаяся ранее в литературе.

5. Предложенное решение

5.1. Архитектура решения

5.1.1. Общая архитектура

Кадровый граф в первую очередь состоит из вершин, каждая из которых в свою очередь состоит из имени и функций *объявления* и *запуска*, задаваемых пользователем рантайма. Как следует из названия, функция запуска вызывается рантаймом в момент исполнения кадрового графа для записи командных списков, а функция объявления вызывается при перекомпиляции графа чтобы предоставить пользователю возможность установить следующие скрытые свойства вершины:

— ресурсные и вершинные зависимости,
— назначения ресурсов,
— режим *мультиплексирования* вершины,
— флаг наличия побочных эффектов,
— требуемое для запуска вершины состояние драйвера.

Пользователь может в любой момент создать, удалить или изменить вершину, на что рантайм автоматически отреагирует на ближайшем запуске кадрового графа. Все существующие в программе вершины организуются в единую структуру называемую кадровым графом, но в строгом математическом смысле кадровым графом не являющуюся.

Каждый кадр приложения пользователь даёт команду рантайму запустить текущий граф. Перед тем как исполнить эту команду, рантайм проверяет, не изменился ли граф с прошлого запуска, и в случае изменения начинает процесс *компиляции графа*, состоящий из следующих пунктов:

1) объявление вершин,
2) генерация промежуточного представления,
3) построение расписания вершин,

4) построение расписания ресурсов,

5) активация *истории ресурсов*.

Помимо вершин, кадровый граф содержит в себе некоторую вспомогательную информацию, при изменении которой нет нужды полностью перекомпилировать граф, поэтому процесс компиляции является инкрементальным. Так, например, при изменении разрешения окна, процесс перекомпиляции графа начнётся с предпоследнего пункта. Первый этап компиляции же запускается всякий раз, когда пользователь изменил множество вершин, и именно в рамках этого этапа запускаются функции объявления изменившихся вершин.

Как уже было сказано выше, данные, изначально задаваемые пользователем рантайма, строго говоря, не являются графом, а лишь схемой получения графа *промежуточного представления*, с которым и работает остальной рантайм. Изначальные данные могут быть невалидными и противоречивыми, поэтому в процессе получения промежуточного представления также идёт валидация, и гарантируется, что получившийся граф будет удовлетворять следующему определению. *Граф промежуточного представления* — пятёрка (V, E, R, U, H), где (V, E) — вершины и рёбра ориентированного ациклического графа, R — множество ресурсов, функция $U : V \to 2^R$ задаёт множество используемых каждой вершиной ресурсов, а функция $H : V \to 2^R$ задаёт множество ресурсов, чью историю использует вершина. При этом требуется, чтобы для любого $r \in R$ подграф (V, E), индуцированный множеством $\{v \in V \mid r \in U(v)\}$, был непустым слабосвязным графом с ровно одной вершиной с входной степенью 0.

На этапе построения расписания вершин строится топологическая сортировка промежуточного графа. В теории, именно на этом этапе может строиться многопоточное расписание запуска функций исполнения вершин, а также могут выбираться разные очереди GPU для отправки команд исполняемы-

ми вершинами. Однако на практике, работа в этом направлении требует крупных предварительных вложений времени в сугубо инженерные вопросы, что не было предусмотрено в рамках бюджета данной работы.

Эти и оставшиеся этапы, а также конкретные возможности разработанного решения, более подробно рассмотрены в следующих подразделах.

5.1.2. Зависимости вершин

Под ресурсами и вершинами как элементами множеств тут и в дальнейшем будем понимать их уникальные строковые идентификаторы, задаваемые пользователем рантайма. У каждой вершины v, среди прочих, есть следующий набор скрытых свойств, устанавливаемых через функцию декларации:

— множества предшествующих и последующих вершин P_v и F_v,

— множество создаваемых ресурсов C_v,

— множество читаемых ресурсов R_v,

— множество изменяемых ресурсов M_v,

— множество пар переименования ресурсов E_v.

Именно на основании этих свойств рантайм генерирует рёбра в промежуточном графе по следующим правилам. Ребро из вершины v в вершину u проводится тогда и только тогда, когда верно хотя бы одно из следующих утверждений:

— $v \in P_u$,

— $u \in F_v$,

— $r \in C_v$ и $r \in M_u$,

— $r \in M_v$ и $r \in R_u$,

— $r \in R_v$ и $\exists r',\ (r, r') \in E_u$,

— $\exists r',\ (r', r) \in E_v$ и $r \in M_u$.

Иначе говоря, рантайм гарантирует, что создание ресурса произойдёт раньше, чем все модификации, каждая из модификаций произойдёт раньше, чем

каждое чтение, и наконец каждое чтение произойдёт раньше, чем переименование (см. рис. 1). При этом само переименование считается моментом создания нового ресурса, соответствующего новому имени. Если в процессе генерации промежуточного представления из пользовательских вершин получается граф с циклами, либо какой-то ресурс создаётся более чем одной вершиной (считая переименования), рантайм оповещает пользователя об ошибке и предпринимает самостоятельную попытку исправить итоговый граф путём игнорирования некоторых вершин или рёбер.

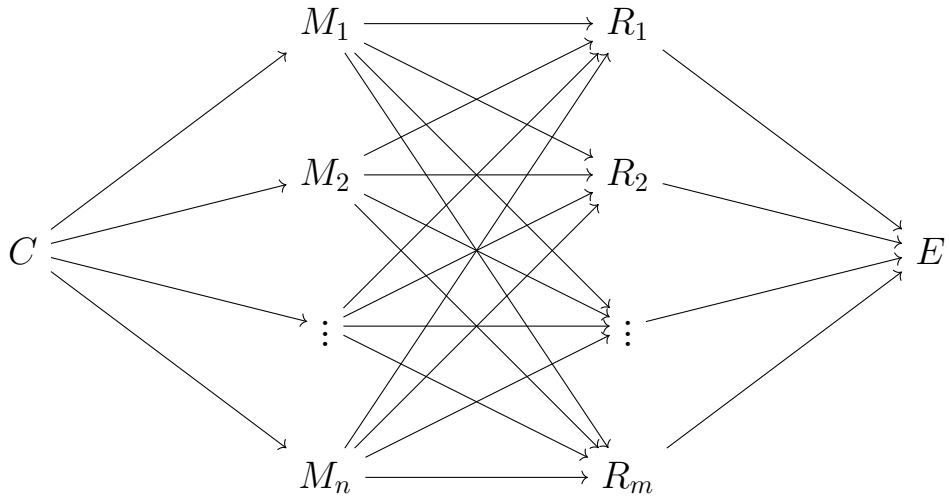

Рис. 1: Визуализация зависимостей между вершинами совершающими различные операции над ресурсом. Здесь, C — вершина, создающая ресурс, M_1, …, M_n — вершины, модифицирующие ресурс, R_1, …, R_m — вершины, читающие ресурс, а E — вершина, совершающая переименование этого ресурса.

При генерации промежуточного представления, информация о переименовывании стирается. Так, одному ресурсу из множества R промежуточного графа может соответствовать несколько различных имён ресурсов, указанных пользователем.

Подобная система упорядочивания гарантирует расширяемость и модульность приложений. Основной компонент приложения может составить *скелетный* кадровый граф, не отображающий ничего, но задающий базовую

структуру рендеринга. 3.схемка? Далее, различные модули приложения могут встраиваться в этот скелет, читая и модифицируя различные ресурсы.

5.1.3. История ресурсов

В процессе интеграции одной из ранних версий рантайма кадровых графов было выяснено, что огромное количество алгоритмов визуализации в целевом приложении использует понятие *истории ресурса*: требуется чтение данных конкретного ресурса в том виде, в котором они находились на конец предыдущего кадра.

Для поддержки таких алгоритмов было введено разделение на *логические* и *физические* ресурсы, где первые представляют собой строковые имена задаваемые пользователем рантайма, а последние — регионы памяти GPU, с которыми работают функции запуска вершин. Для каждого логического ресурса кадрового графа создаётся два физических ресурса, предоставляемых пользователю поочерёдно на чётных и нечётных кадрах. Это позволяет вершинам, представляющим такие алгоритмы как темпоральное сглаживание [24], одновременно читать историю ресурса и писать сам ресурс.

В дополнение вышеперечисленным множествам запрашиваемых ресурсов каждая вершина содержит множество H_v ресурсов, чья история требуется для выполнения вершины. Запрос истории с точки зрения упорядочивания вершин приравнивается к чтению, что делает невозможным запрос истории ресурсов, переименуемых в процессе исполнения кадра.

Насколько известно автору, разработанное решение — первый рантайм кадровых графов, поддерживающий запросы истории ресурсов, а также способный переиспользовать память ресурсов, чья история запрашивается хотя бы одной вершиной.

5.1.4. Мультиплексирование

Следующая проблема, возникшая в процессе интеграции рантайма — необходимость запускать некоторые вершины кадрового графа несколько раз в рамках одного кадра. Возникает такая необходимость в следующих ситуациях:

— запуск приложения на устройстве виртуальной реальности, где вид с основной камеры необходимо рендерить для каждого дисплея заново (от 2 до 4 в зависимости от устройства);

— поддержка скриншотов в высоком разрешении, где скриншот рендерится по-частям, чтобы не превысить бюджет видеопамяти потребительских GPU;

— поддержка алгоритма «SSAA» [25];

— поддержка локальной многопользовательской игры, где экран делится пополам, и на разных половинах отображается ракурс разных игроков.

Для поддержки этих ситуаций в предлагаемом решении был введён механизм *мультиплексирования* вершин и ресурсов.

Вводится размерность мультиплексирования, D. Каждая вершина посредством функции декларации указывает свой *режим мультиплексирования*, элемент $\mathbb{Z}_2^D$, являющийся булевой маской выбора измерений. На режимах мультиплексирования вводится частичный порядок: для $a, b \in \mathbb{Z}_2^D$, $a \preceq b$, если $\forall i,\ a_i \leqslant b_i$. Каждому ресурсу ставится в соответствие режим мультиплексирования той вершины, которая его создаёт. Если вершина с режимом a запрашивает ресурс с режимом b, то от пользователя требуется, чтобы $b \preceq a$, так как иная ситуация ведёт к неоднозначности выбора физического ресурса. Также при $u \in P_v$ или $u \in F_v$ требуется, чтобы режимы u и v были сравнимы в частичном порядке $\preceq$.

Перед запуском графа, от пользователя требуется предоставить рантайму $c \in \mathbb{Z}_{>0}^D$. Во время генерации промежуточного представления, для верши-

ны с режимом a будет создано $\prod_{i=1}^{D} c_i^{a_i}$ вершин промежуточных промежуточного графа, и аналогично для ресурсов. Правила проведения рёбер в промежуточном графе обобщаются со случая $c = (1, \ldots, 1)$ естественным образом в силу ограничения из предыдущего абзаца.

Предлагаемая система мультиплексирования покрывает все описанные выше случаи. Рассмотрим, например, приложение, работающее на VR-устройстве с двумя дисплеями, и использующее алгоритмом сглаживания SSAA x4 Выберем $D = 2$ и $c = (2, 4)$, где первое измерение будет отвечать номеру дисплея, а второе — количеству суперпикселей. Все вершины приложения, визуализирующие вид с основной камеры, должны быть помечены режимом $(1, 1)$. Вершины же, например, вычисляющие тени, должны быть помечены режимом $(0, 0)$, так как одна и та же карта теней может быть использована для обоих глаз и всех суперпикселей.

5.1.5. Автоматические разрешения текстур

Большие неудобства при разработке графических приложений составляет смена разрешения экрана. Большинство транзиентных ресурсов представляют собой текстуры, разрешение которых кратно разрешению окна приложения, что делает целесообразным разработку централизованного механизма реакции на смену разрешения экрана.

Таким механизмом в предлагаемом решении являются *авторазрешения*. В вершине, создающей транзиентную текстуру, пользователь рантайма вместо конкретного разрешения может указать строковой идентификатор авторазрешения, числовое значение соответствующее которому сообщается рантайму извне, как правило, в коде реакции на смену разрешения окна. Рантайм, обнаружив на очередном кадре, что одно из авторазрешений поменялось, построит новое расписание ресурсов, тем самым изменяя разрешения физических ресурсов прозрачно для пользователя.

Также система авторазрешений предлагаемого решения поддерживает *динамическую смену разрешений*. Обнаружив низкую частоту кадров, приложение может уменьшить разрешение, в котором рисуются кадры, чтобы увеличить производительность. Рантайм кадровых графов в свою очередь позволяет уменьшать разрешения всех текстур с производительностью достаточной, чтобы делать это каждый кадр, не освобождая при этом, однако, неиспользуемую память.

5.1.6. Расстановка барьеров

Как уже было упомянуто ранее, современные графические API требуют от пользователя в ручную отслеживать использование ресурсов и расставлять в соответствии со спецификацией необходимые ресурсные барьеры.

В разработанном решении для каждого ресурса, встречающегося во множествах C_v, R_v, M_v, E_v и H_v, в вершине хранится пометка о том, как именно в рамках её функции запуска будет использоваться ресурс Эта информация сохраняется при переходе к промежуточному представлению, и в момент построения расписания ресурсов рантайм в соответствии с этими пометками расставляет так называемые *раздельные барьеры* [6, раздел 7.5], позволяющие GPU самостоятельно выбирать подходящий момент времени для выполнения служебной работы по управлению кешами и состояниями ресурсов.

Как будет видно в последующих разделах, необходимость указывать способ использования каждого ресурса в каждой вершине не доставляет больших неудобств пользователям благодаря разработанному публичному API рантайма.

5.1.7. Однородная поддержка ресурсов CPU

В начале работы над рантаймом планировалась поддержка лишь GPU ресурсов, однако в процессе интеграции первого прототипа было выяснено,

что передача CPU данных, влияющих на запись командных списков, должна тесно взаимодействовать с различными подсистемами рантайма. Вследствие этого было принято решение обобщить механизмы кадровых графов на работу как с данными на GPU, так и с данными на CPU. Это решение способствует модульности и простоте написания кода новых вершин. Так, например, матрица преобразования из мирового пространства в экранное пространство должна отличатся для разных глаз в VR-приложениях. Чтение этой матрицы во всех вершинах через ресурс кадрового графа позволяет подменить эту матрицу в базовом скелете кадрового графа при адаптации приложения для VR-платформ, не меняя при этом код других вершин.

5.1.8. Прореживание и валидация кадрового графа

Как уже было упомянуто выше, в процессе генерации промежуточного представления происходит валидация корректности данных пользователя.

Так, например, возможна ситуация, в которой вершина запрашивает на чтение не произведённый никакой другой вершиной ресурс. Чтобы предотвратить падения приложения в подобных ситуациях, рантайм находит и исключает из рассмотрения подобные *сломанные* вершины. Так как сломанная вершина сама могла создавать некоторые ресурсы, в результате этого исключения могут появиться новые сломанные вершины, поэтому процедура запускается итеративно, пока находится хотя бы одна сломанная вершина. Однако встречаются ситуации, в которых вершина способна функционировать, даже если один из запрашиваемых ресурсов отсутствует. Например, некоторые алгоритмы способны работать с глубиной в различных форматах, предоставляя, однако, разный по качеству результат. Для учёта таких ситуаций запросы ресурсов в вершине могут быть помечены как опциональные, что предотвратит исключение вершины из графа в случае отсутствия соответствующего ресурса.

Возможна и обратная ситуация: запуск вершины может не иметь никаких побочных эффектов кроме создания ресурса, который в итоге не будет использован ни одной другой вершиной. Чтобы исключить трату вычислительных ресурсов и памяти подобными *висячими* вершинами, рантайм находит и исключает из рассмотрения и их. Однако в процессе интеграции рантайма кадровых графов в движок не редко возникали ситуации, в которых побочные эффекты запуска вершины производились незаметно для кадрового графа, в обход системы ресурсов. Чтобы подобные вершины не выкидывались рантаймом, функция декларации может пометить вершину флагом внешних побочных эффектов.

5.1.9. Внешние ресурсы

Некоторые особые виды ресурсов не могут управляться рантаймом кадровых графов, так как само графическое API не предоставляет прямой доступ к ним. Например, изображения *свопчейна* создаются операционной системой, и хранятся в памяти управляемой ей. Более того, для интеграции некоторых часто используемых ресурсов движка в кадровый граф требовалось слишком большое количество изменений в коде. Для постепенной интеграции этих ресурсов, а также поддержки внешних по отношению ко всему приложению ресурсов, в рантайм было добавлено понятие *внешних ресурсов*. Внешние ресурсы работают аналогично остальным за исключением управления их памятью. При создании внешнего ресурса, вместо предоставления данных, необходимых для создания подобного ресурса в рамках рантайма, от пользователя требуется предоставить функцию, отображающую текущий *индекс мультиплексирования* в уже существующий физический ресурс. Здесь, индекс мультиплексирования — вектор, описывающий номер запуска вершины в рамках мультиплексирования. Рантайм в свою очередь запускает эту функцию перед каждым запуском соответствующий вершины, и предоставляет всем последу-

ющим вершинам полученный физический ресурс в качестве запрошенного логического.

5.1.10. Управление глобальным состоянием

Как правило, перед запуском работы на GPU посредством вызова отрисовки или запуска вычислительного шейдера, необходимо выставить некоторый набор глобальных состояний GPU посредством специальных команд. Для нескольких подряд идущих вершин, эти команды могут быть абсолютно одинаковыми. Так, например, при отрисовке G-буфера, набор целевых изображений не будет отличатся между вершинами. Переключение этих состояний — не самая долгая операция, однако переключая их зазря перед каждым вызовом отрисовки всё равно можно значительно замедлить работу приложения.

Чтобы избежать этой ситуации, выставление состояний GPU было интегрировано в рантайм кадровых графов. Пользователь указывает необходимое на момент запуска состояние в функции декларации, а рантайм находит оптимальную последовательность смены состояний в процессе исполнения графа, минимизируя тем самым накладные расходы.

5.2. Пользовательское API рантайма

В рамках данной работы также было разработано эргономичное для разработчиков API для работы с рантаймом кадровых графов. Рассмотрим простейший пример его использования, изображённый на рисунке 2. Вершины создаются при помощи функции `dabfg::register_node` (строка 3), возвращающей объект `dabfg::NodeHandle`. Созданная вершина автоматически попадает в глобальный кадровый граф и автоматически удаляется, как только удалён соответствующий объект `dabfg::NodeHandle`. На вход функция регистра-

```
1  dabfg::NodeHandle makeExampleNode1()
2  {
3    return dabfg::register_node("example_node_1", DABFG_PP_NODE_SRC,
4      [](dabfg::Registry registry)
5      {
6        auto cpuDataHndl = registry
7          .readBlob<DataType>("cpu_data")
8          .handle();
9
10       return
11         [cpuDataHndl]()
12         {
13           const DataType &data = *cpuDataHndl;
14           …
15         };
16     });
17 }
```

Рис. 2: Листинг элементарного примера использования разработанного API.

ция вершин принимает название, специальный макрос[2] и лямбда-функцию декларации (строка 4), от которой в свою очередь требуется вернуть функцию запуска вершины (строка 11). Подобный дизайн вложенных лямбд позволяет захватывать *хэндлы* ресурсов, получаемые в функции декларации, и получать из них данные в функции запуска. Так, например, в строке 6 у объекта `dabfg::Registry`, отвечающего за декларативное описание скрытых параметров вершины, запрашивается на чтение CPU-ресурс `"cpu_data"` с типом `DataType`. Далее, хэндл этого ресурса захватывается в лямбда-функцию запуска, и разыменовывается для доступа к соответствующим данным, созданным в иной вершине. Стоит отметить, что доступ к данным предоставляется по константной ссылке, только на чтение, так как для изначального запроса был использован шаблон функции `readBlob`. В случае использования парного шаблона `modifyBlob`, обозначающего модификацию ресурса и соответственно ина-

[2]Использование макроса необходимо для получения названия файла и номера текущей строки, используемых для диагностики кадровых графов.

че влияющего на упорядочивание вершин, тип возвращаемого хэндла меняется и предоставляет доступ и на запись. Наконец, режимом мультиплексирования по-умолчанию для вершин является полное мультиплексирование, поэтому функция запуска будет запущена не единожды, и более того, на каждом запуске разыменовывание `cpuDataHandle` будет возвращать ссылку на разные объекты в памяти, соответствующие разным индексам мультиплексирования.

Рассмотрим более сложный пример, использующий всю силу разработанного API. На рисунке 3 приведён листинг вершины, рисующей некоторый набор объектов, получаемый через CPU-ресурс `"objects"` при помощи предположительного шейдера `"shaderForObjects"`[3]. Однако известно, что перед тем как запускать отрисовку шейдером, необходимо выставить различные состояния драйвера. В данном примере все необходимые состояния выставляются полностью через разработаннео API. В строке 12 запрашивается *виртуальный проход отрисовки*, в рамках которого целями для отрисовки цветов выбраны логические ресурсы `"color"` и `"additional_color"`, а буфером глубины ресурс `"gbuf_depth"`. Стоит заметить отличие в способах указания этих ресурсов в качестве целей отрисовки. Для глубины использован стандартный полный синтаксис: явно запрошен соответствующий ресурс глубины на модификацию, и затем объект запроса указан в качестве глубины прохода отрисовки. Для `"additional_color"` же использован сокращённый синтаксис: указание названия ресурса напрямую в функциях `color` и `depth` автоматически запрашивает соответствующий ресурс на модификацию. Для `"color"` же использовано переименование, влияющие описанным в предыдущих разделах образом на порядок исполнения вершин и также считающееся модификацией с точки зрения использования ресурсов в рамках запуска вершины. Более того, в строке 9 указано, что у переименованного ресурса

[3]Функция `createShader` использована исключительно для экспозиции и не является частью разработанного API.

```
dabfg::NodeHandle makeExampleNode2(bool useHistory)
{
  return dabfg::register_node("example_node_2", DABFG_PP_NODE_SRC,
    [useHistory](dabfg::Registry registry)
    {
      registry.orderMeAfter("some_other_node");

      auto colorRequest = registry.renameTexture("color",
        "color_after_example", dabfg::History::ClearOnFirstFrame);
      auto depthRequest = registry.modifyTexture("gbuf_depth");

      registry.requestRenderPass()
        .color({std::move(colorRequest), "additional_color"})
        .depth(std::move(explicitDepthRequest));

      registry.readTexture("some_optional_input_texture")
        .optional()
        .atStage(dabfg::Stage::PS)
        .bindToShaderVar("some_optional_shader_input");

      if (useHistory)
        registry.readTextureHistory("color_after_example")
          .atStage(dabfg::Stage::PS)
          .bindToShaderVar("previous_frame_color");

      auto objectsHndl = registry
        .readBlob<ObjectContainer>("objects")
        .handle();

      return
        [objectsHndl, shader = createShader("shaderForObjects")]()
        {
          shader.render(*objectsHndl);
        };
    });
}
```

Рис. 3: Листинг усложнённого примера использования разработанного API.

должна быть включена история, а также указано, что на первом кадре после компиляции графа в качестве истории рантайм должен предоставить очищенную текстуру. В строке 21 же в зависимости от настройки, приходящей извне, запрашивается история ресурса, созданного в результате переименования, `"color_after_example"`, и сразу же привязывается в качестве значения входного параметра `"previous_frame_additional_color"` пиксельного шейдера. Заметим, что указание набора шейдеров в которых будет использоваться текстура в качестве семплируемого параметра обязательно для корректной расстановки барьеров. В случае использования ресурса как цели прохода отрисовки же ничего указывать не требуется, так как подобное использование возможно лишь в пиксельном шейдере. Отметим также, что в случае использования GPU ресурса напрямую в функции посредством захвата его хэндла, абсолютно необходимо указать способ его дальнейшего использования при помощи функций `atStage` и `useAs`. В строке 16 же запрашивается `опциональный` ресурс, в случае отсутствия которого вершина всё равно будет запущена, а в соответствующий входной параметр шейдера будет помещена пометка об отсутствии текстуры. Наконец, в строке 6 текущая вершина явно упорядочивается с другой вершиной. Однако механизм прямого упорядочивания вершин менее устойчив к изменению и модуляризации кода, поэтому предполагается к использованию лишь в крайних случаях и в качестве временного решения в процессе миграции кодовой базы на кадровый граф.

В предлагаемом API есть ряд инвариантов корректности. Например, необходимо, чтобы пользователь не мог получить хэндл GPU-ресурса, не указав как конкретно он собирается его использовать; недопустимо, чтобы в качестве цветовой цели прохода отрисовки был указан запрос на чтение ресурса; а также перед вызовом функции `bindToShaderVar` необходимо указывать вид шейдера, в котором ресурс будет использоваться. При помощи шаблонов языка C++ в разработанном API были достигнуты гарантии соблюдения

этих многих из этих инвариантов на этапе компиляции. Фактически каждая функция, вызываемая в рамках функции декларации, возвращает объект нового типа. Так, у типа представляющего запрос ресурса до вызова функции `atStage` полностью отсутствует метод `bindToShaderVar`, а вызов `atStage` возвращает объект нового типа, у которого уже есть этот метод. Аналогично различными типами обладают объекты, возвращаемые из функций `readTexture`, `modifyTexture`, `readBlob` и так далее, обладающие лишь необходимым функционалом. Подобный дизайн API позволяет избежать многих элементарных ошибок в рамках разработки приложения, основанного на кадровом графе, экономя время и силы прикладных разработчиков.

5.3. Построение расписания ресурсов

Последний и наиболее важный в данной работе этап компиляции кадрового графа — построение расписания ресурсов. На вход этому этапу поступает промежуточное представление (V, E, R, U, H), а в результате необходимо построить отображение $\Omega : i, j, k \mapsto \rho$, где i — логический ресурс, j — номер кадра по модулю 2, k — индекс мультиплексирования, а ρ — физический ресурс. Это отображение и будет использоваться перед запуском вершин рантаймом, чтобы предоставить пользователю физические ресурсы для чтения и записи. Более того, именно на данном этапе рантайм вычисляет *события*, происходящие с ресурсами: активация, деактивация и барьеры. Во время исполнения кадрового графа на основе этих событий в командные списки записываются соответствующие команды между запуском вершин. Этот процесс не представляет научного интереса, поэтому оставшаяся часть раздела посвящена получению Ω.

Для дальнейшего построения алгоритма потребуется ввести понятие циклического времени. Так, в дальнейшем, термин *время* будет подразумевать элементы циклической группы Z_T, где $T = 2\,|V|$. Для $a, b \in Z_T$, обозна-

чим за $[a, b)$ множество $\{a, a+1, \ldots, b\}$ если $a < b$, и множество $\{b, b+1, \ldots, T-1\}$ $\{0, 1, \ldots, a\}$ иначе. Предположим, что вершины промежуточного графа V уже прономерованы в порядке выбранной топологической сортировки. Определим времена жизни физических ресурсов, порождённых ресурсом i, $[l_i^e, r_i^e)$ и $[l_i^o, r_i^o)$:

$$l_i = 1 + \min_{i \in U(v_j)} j$$

$$r_i = 1 + \max_{i \in U(v_j)} j$$

$$r'_i = 1 + \max_{i \in H(v_j)} j$$

$$l_i^e = l_i,$$

$$l_i^o = |V| + l_i,$$

$$r_i^e = \begin{cases} |V| + r'_i, & r'_i \text{ определено} \\ r_i, & \text{иначе} \end{cases}$$

$$r_i^o = \begin{cases} r'_i, & r'_i \text{ определено} \\ |V| + r_i, & \text{иначе} \end{cases}$$

где r'_i не определено тогда и только тогда, когда история ресурса i ни разу не была использована; см. рис. 4. Далее, для полученных интервалов времени жизни физических ресурсов и размеров соответствующих логических ресурсов, необходимо решить задачу дискретной оптимизации, называемую в рамках данной работы циклической динамической аллокацией ресурсов (от англ. cyclic dynamic storage allocation, сокращённо CDSA), и формулируемую следующим образом.

Пусть $l_i, r_i \in \mathbb{Z}_T$, $s_i \in \mathbb{Z}_{>0}$, $i = \overline{0, n}$. Допустимой функцией аллокации называют функцию $\alpha : \overline{0, n} \to \mathbb{Z}_{\geqslant 0}$ такую, что для любых $i \neq j$, либо $[l_i, r_i) \cap [l_j, r_j) = \emptyset$ в смысле циклического времени, либо $[\alpha(i), \alpha(i) + s_i) \cap$

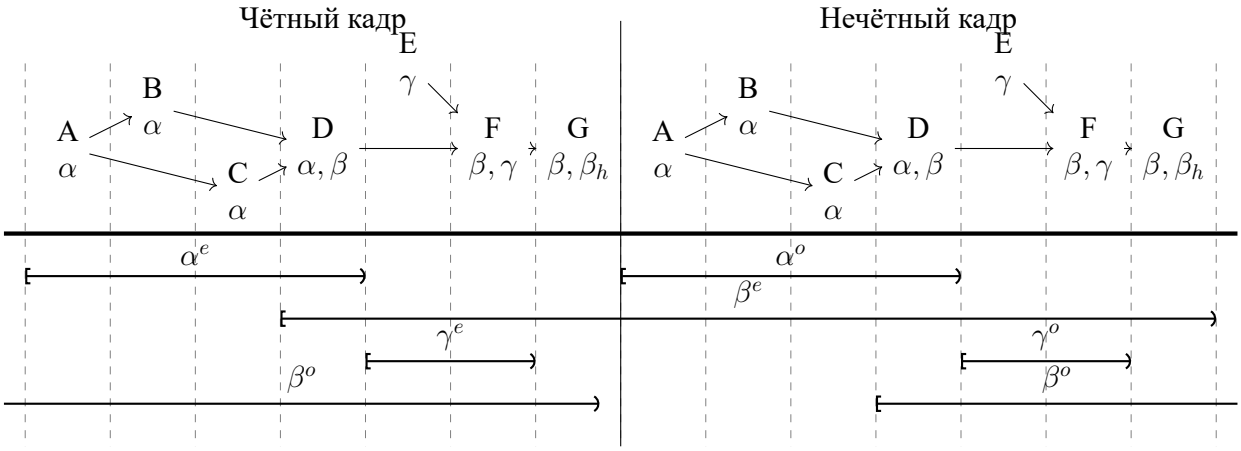

Рис. 4: Визуализация отрезков времени жизни физических ресурсов. В верхней половине изображён промежуточный граф с выбранным порядком исполнения. Вершины, обозначенные заглавными латинскими буквами, исполняются слева направо. Под вершинами перечислены списки запрашиваемых ими ресурсов, обозначенных греческими буквами, а также через нижний индекс h обозначены запрашиваемые истории ресурсов. В нижней половине изображены отрезки времён жизни физических ресурсов для чётных и нечётных кадров соответственно.

$[\alpha(j), \alpha(j) + s_j) = \emptyset$. Для функции аллокации α, определена величина $makespan = \max_i \alpha(i) + s_i$. Задача — найти допустимую функцию аллокации $\alpha_{\min}$, минимизирующую величину $makespan$.

Эта задача — обобщение хорошо известной в литературе задачи динамической аллокации памяти (от англ. dynamic storage allocation, сокращённо DSA), необходимое для учёта запросов к истории ресурсов. Рассмотрим частный случай этой задачи, в котором все веса ресурсов единичны, то есть $\forall i,\ s_i = 1$. Ясно, что решение этого частного случая эквивалентно раскраске графа пересечений множеств $[l_i, r_i)$, которые в свою очередь эквивалентны дугам окружности длины T. Известно, что задача раскраски графа пересечений дуг окружности NP-полна [26]. Следовательно, и поставленная задача CDSA NP-полна, а значит для её решения необходимы схемы приближения.

В рамках данной работы предлагается следующий алгоритм приближённого решения задачи CDSA, см. алгоритм 1. Вход алгоритма — множество ресурсов, троек (l_i, r_i, s_i), выход — функция α, хранимая в виде массива, и обозначающая сдвиг каждого ресурса относительно начала *кучи*, последовательности страниц видеопамяти. В рамках алгоритма *живым* в момент времени t ресурсом называют такой ресурс i, что $t \in [l_i, r_i)$. В процессе работы алгоритма поддерживаются 2 структуры: множество живых в текущий момент времени t ресурсов Y с пометками $until$ соответствующих блоков и множество уже однажды аллоцированных но доступных для переиспользования блоков памяти A, где блок — тройка $(offset, size, until)$, позиции в рамках кучи, размер блока и время, до которого этот блок доступен, соответственно. В программной реализации для хранения этих структур используются красночёрные деревья поиска. В некоторые моменты алгоритма происходит *дефрагментация* множества A: блоки, имеющие одинаковую пометку $until$ и расположенные в памяти подряд, объединяются в один большой блок. Далее, также в рамках алгоритма, а также дальнейшего анализа, используется величина $L(t) = \sum_{i:t\in[l_i,r_i)} s_i$, называемая *нагрузкой* в момент времени t. В начале алгоритма, в строке 5, выбирается опорная точка t_0 — начало движения сканирующей прямой. В дальнейшем текущее положение сканирующей прямой хранится в переменной t, а переменная H используется для отслеживания текущего размера кучи. В строках 7–12 в память подряд укладываются ресурсы, живые в опорной точке t_0, а также добавляются во множество Y с началами их времён жизни в качестве пометок. Далее, алгоритм повторяет цикл на строках 13–37 пока не кончатся элементы в X, выбирая каждый раз ресурс с началом времени жизни, ближайшим к текущему положению сканирующей прямой t, что в программной реализации достигается путём сортировки массива X относительно t_0. В рамках этого цикла в первую очередь, в строках 14–20, «освобождаются» ресурсы, чьё время жизни уже закончилось, возвращая для них соответствующие блоки памяти в A. Следующим шагом

Алгоритм 1 Предлагаемый жадный алгоритм решения CDSA

$X \leftarrow$ входное множество ресурсов
$Y \leftarrow \emptyset$ — мн-во пар живых ресурсов и пометок $until$ соответствующих блоков
$A \leftarrow \emptyset$ — мн-во доступных блоков $(offset, size, until)$
$H \leftarrow 0$ — текущий размер кучи
$t_0 \leftarrow$ момент времени с наименьшим $L(t)$
$t \leftarrow t_0$
for ресурс i жив в t **do**
    $\alpha(i) \leftarrow H$
    $H \leftarrow H + s_i$
    Удалить i из X
    Добавить (i, l_i) в Y
end for
repeat
    for $(j, until) \in Y$ **do**
        if j не жив в t **then**
            Удалить j из Y
            Добавить $(\alpha(j), s_j, until)$ в A
            Дефрагментировать A
        end if
    end for
    $i \leftarrow$ элемент X с наименьшим $l_i - t \mod T$
    Удалить i из X
    if выбор эл-та A провалится на следующем шаге **then**
        Добавить блок (H, s_i, ∞) в A
        Дефрагментировать A
        $H \leftarrow H + s_i$
    end if
    $a \leftarrow$ блок с наименьшим $size \geqslant s_i$ в A такой, что $[l_i, r_i) \cap [until, t_0) = \emptyset$ или $until = \infty$
    Удалить a из A
    $\alpha(i) \leftarrow a.offset$
    if $a.size > s_i$ **then**
        Добавить $(a.offset + s_i, a.size - s_i, a.until)$ в A

алгоритм выбирает новый ресурс в строке 21, помечает его живым, и пытается переиспользовать уже имеющийся в A блок памяти. Впрочем, подходящего блока может не найтись. В таком случае в строках 23–27 создаётся новый блок подходящего размера на вершине кучи памяти, с пометкой $until = \infty$. После этого в строке 28 находится блок согласно стратегии «best-fit», то есть блок минимального размера среди тех, в которых помещается блок, и при этом время жизни ресурса не пересекает границу $until$. Ровно для последнего условия в рамках алгоритма отслеживаются эти пометки, что гарантирует отсутствие пересечений с выделенными в строках $7 - -12$ ресурсами. Позиция в памяти выбранного блока берётся в качестве значением функции α для текущего ресурса. Однако выбранный блок мог быть много больше по размеру, чем ресурс, поэтому оставшееся в нём свободное место отрезается и добавляется обратно в структуру A. Наконец, позиция сканирующей прямой обновляется до начала текущего ресурса, l_i, и алгоритм переходит к следующей итерации.

Получив функцию аллокации и размер итоговой кучи, несложно построить итоговую для данного этапа компиляции кадрового графа функции Ω. При помощи графического API создаётся куча видеопамяти соответствующего размера, а затем для каждого номера физического ресурса p на позиции $\alpha(p)$ в куче создаётся соответствующий ресурс. Далее, для логического ресурса i, номера кадра j по модулю 2 и индекса мультиплексирования k, найдём соответствующий номер физического ресурса p, и определим $\Omega(i, j, k)$ как созданный ресурс графического API, соответствующий индексу p.

В заключение данного раздела отметим, что на практике некоторые платформы имеют несколько различных видов видеопамяти. Программная реализация предложенного метода поддерживает эту особенность, группируя физические ресурсы по целевым типам видеопамяти, решая для каждого типа памяти задачу CDSA независимо и соответственно создавая несколько

независимых куч.

6. Результаты

В процессе выполнения данной работы в рамках движка Dagor было реализовано предложенное решение в рамках библиотеки рантайма кадровых графов «daBFG», занимающей 13828 строк кода на языке C++, а также была произведена частичная интеграция этой библиотеки в существующие проекты, основанные на этом движке. Так как масштаб кода отрисовки этих проектов невероятно велик, интеграция производится итеративно. В результате изначальной интеграции, проведённой автором работы, было выделено около 30 вершин и порядка 10 ресурсов. С тех пор к усилиям по интеграции присоединились и другие разработчики, и на данный момент в проекте Enlisted на максимальных настройках кадровый граф состоит из 84 вершин и отслеживает 74 ресурса, 27 из которых расположены в видеопамяти, управляемой самим рантаймом (остальные — CPU-ресурсы и внешний ресурсы). Отметим, что более чем у половины из имеющихся ресурсов запрашивается история хотя бы одной вершиной. Также стоит заметить, что некоторые проекты поддерживают работу в VR, что удваивает количество физических ресурсов, а также один из проектов поддерживает совершение скриншотов в высоком качестве, что за счёт использования SSAA x4 и увеличения разрешения скриншота в 2 раза, посредством мультиплексирования увеличивает количество физических ресурсов в 64 раза. Работа над интеграцией различных частей кода в кадровый граф продолжается, поэтому чтобы оценить качество разработанного решения управления памятью были использованы синтетические тесты.

6.1. Синтетические тесты

В рамках работы над интеграцией решения в Dagor было замечено, что экземпляры задачи CDSA получаемые из временных ресурсов графических приложений обладают определённой структурой. Например, большая часть

таких ресурсов является текстурами, чьё разрешение кратно разрешению пользовательского монитора. Также, может показаться, что запрос истории ресурса — не самая частая операция. Однако, как уже было сказано выше, среди ресурсов, чья память управляется кадровым графом, таких, чья история запрашивается, на данный момент больше половины. Чтобы учесть эту специфику в синтетических тестах, был применён статистический метод *бутстрэпа*.

Метода бутстрэпа заключается в генерации повторной выборки бóльшего размера из эмпирического распределения имеющейся небольшой выборки. В качестве выборки были выбраны следующие параметры: количество моментов времени T, времена жизни ресурсов $|[l_i, r_i)|$ и размеры ресурсов s_i. Также сделано предположение о том, что эти величины независимы от значений l_i. Эмпирическое распределение этих данных было получено из нескольких проектов, основанных на Dagor, на различных настройках качества графики.

В качестве базового метода был выбран распространённый в других решениях подход к обработке запросов истории ресурсов: ресурсы, история которых запрашивается хоть единажды, хранятся отдельно от всех остальных и их память никогда не переиспользуется, а для остальных ресурсов решается задача DSA посредством жадного аллокатора реального времени.

Метрика для сравнения алгоритмов была заимствована из предыдущих работ по приближённому решению задачи DSA, $makespan/LOAD$, где *общая нагрузка* определяется как максимум нагрузок по всем моментам времени, $LOAD = \max_t L(t)$. Эта метрика хорошо подходит для исследования задачи DSA, так как $LOAD$ служит нижней границей для величины $makespan$, хотя часто и недостежимой. Заметим, что в случае CDSA, несложно привести пример последовательности входных данных задачи с увеличивающимся количеством ресурсов такой, что $makespan/LOAD \geqslant 3/2$ для каждого элемента последовательности, см. 4.самоцитирование. На практике, однако, подобные

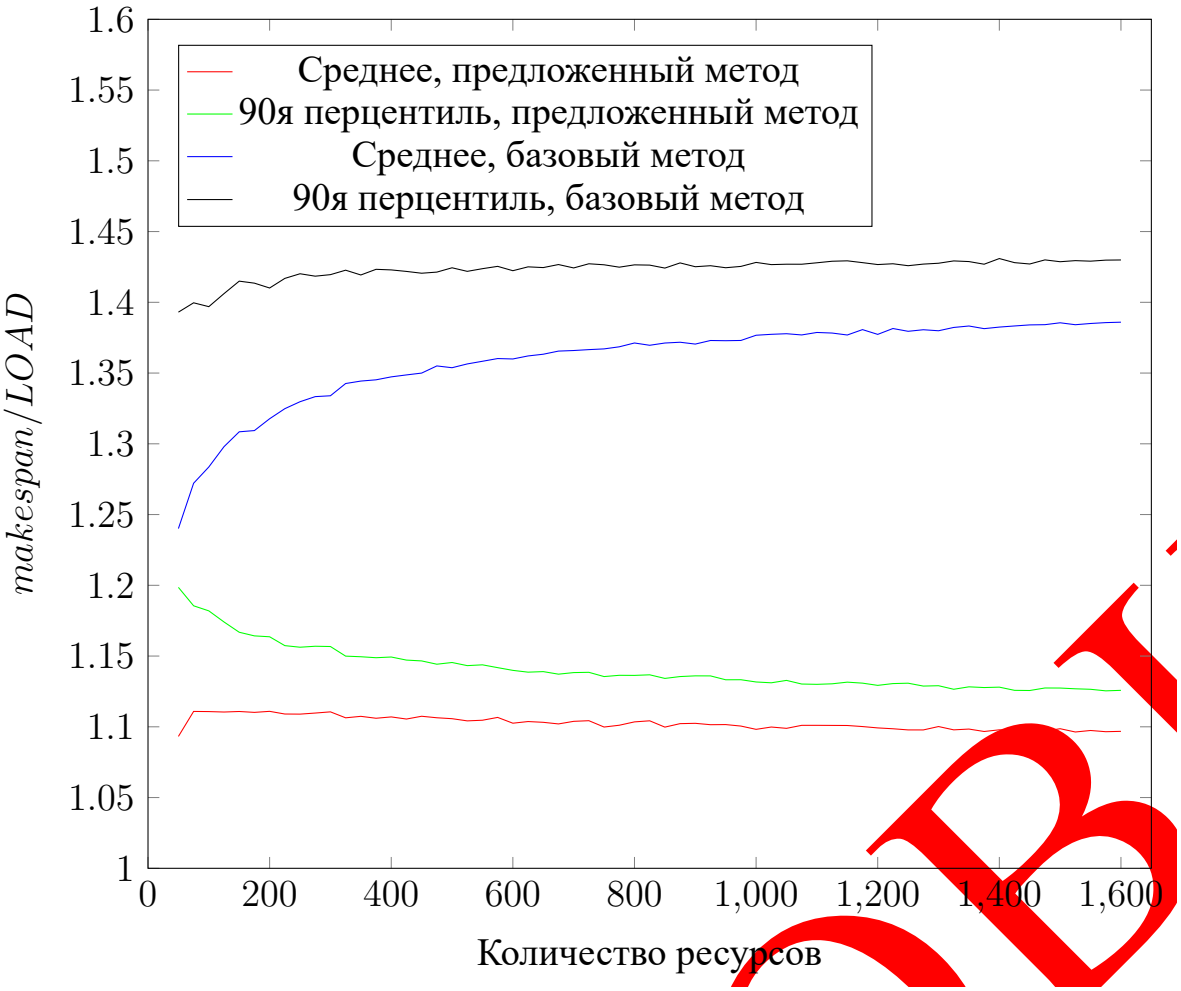

Рис. 5: Замеры качества предложенной схемы решения CDSA по метрике $makespan/LOAD$, меньше — лучше. Значения усреднены по 2000 запускам на синтетических тестах, полученных из реальных данных методом бутстрэпа.

эффекты не возникают, как будет видно из графиков.

Результаты сравнений представлены на рисунке 5 в виде зависимости метрики $makespan/LOAD$ от количества ресурсов, где для каждого количества ресурсов замеры усреднялись по 2000 различным синтетическим тестам. Из рисунка 5 видно, что базовый метод ведёт к росту метрики с количеством ресурсов как в среднем так и в худшем случае. Предложенное решение же приводит к убыванию метрики к асимптоте в районе 1.1, как в среднем, так и в худшем случае, что лучше видно на рисунке 6. Подобный тренд полагает

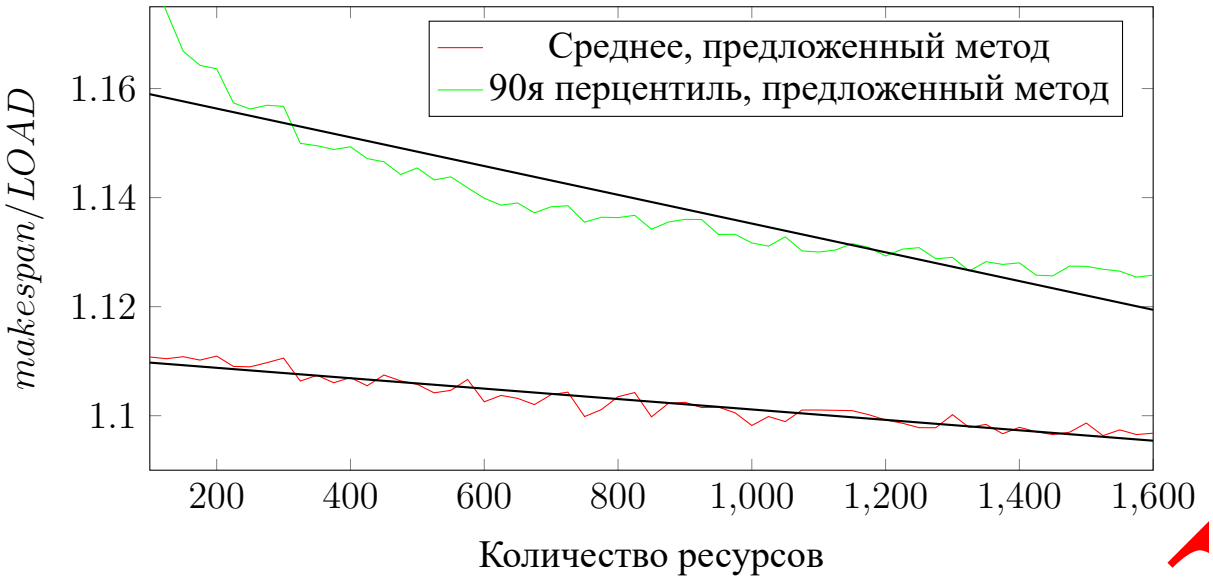

Рис. 6: Графики предлагаемого решения из рис. 5 в увеличенном масштабе.

судить, что чем больше временных ресурсов движка Dagor интегрировано в кадровый граф, тем меньше будет управление их памятью отличатся от оптимального. Оптимальное же отношение $makespan/LOAD$ находится в интервале $[1, 1.1]$, но нахождение этого оптимума не представляется возможным в силу NP-полноты задачи.

Наконец стоит отметить, что предложенное решение очевидно не гарантирует ограниченности метрики $makespan/LOAD$ для последовательности экземпляров задачи CDSA с растущим количеством ресурсов. Иначе говоря, алгоритм имеет неограниченную ошибку относительно оптимального ответа. Достаточно рассмотреть последовательность входов размера n следующего вида, $T = n+2, l_i = i, r_i = i+2, i = \overline{0, n}$, при этом такую, что $\forall j,\ \sum_{i=0}^{j-2} s_i < s_j$ и $s_{j-1} < s_j$. По индукции, при поиске блока для очередного ресурса i будет доступен лишь единственный блок размера $\sum_{j=1}^{i-2} s_j$, по построению не достаточный для размещения ресурса i. Следовательно, ресурсы будут расположены в памяти последовательно, а итоговой ответ $makespan = \sum_{i=0}^{n} s_i$. Однако если рассмотреть эквивалентный экземпляр CDSA, в котором время повёрнуто вспять, алгоритм найдёт решение с $makespan = s_n + s_{n-1} = LOAD$, яв-

ляющееся оптимальным. Отношение $\left(\sum_{j=1}^{i-2} s_j\right) / (s_n + s_{n-1})$ можно сделать сколь угодно большим выбрав подходящее значение s_i, например, 2^{2^i}, а значит и ошибка алгоритма не ограничена. **5.Тут ОООООООООЧЕНЬ вероятна лажа.**

7. Заключение

В рамках данной работы был составлен обзор существующих решения, разработана архитектура рантайма кадровых графов, написана программная реализация предлагаемой архитектуры, произведена интеграция разработанного программного решения в коммерческие проекты, вопрос управления памяти был сведён к не рассмотренной ранее в литературе задаче дискретной оптимизации CDSA, предложена схема приближённого решения этой задачи, и проанализированы качество и скорость работы предложенной схемы. Предложенное решение управления памятью позволяет сэкономить близкое к оптимальному количество памяти на данных, приближенных к реальным.

В рамках дальнейших исследований, во-первых, планируется уделить больше ресурсов проверке гипотезы о влиянии стратегии выбора топологической сортировки на значение $LOAD$ из раздела **??**. Во-вторых, имеет смысл обобщение алгоритмов решения DSA предложенных в работах Йордана Гергова [19; 20] на случай CDSA. В-третьих, крупной, но малоизученной областью развития кадровых графов является автоматическая параллелизация вычислений как на CPU, так и на GPU. Наконец, объединяя все предыдущие пункты, дальнейший интерес представляет возможность адаптации алгоритмов оптимизации кадрового графа под специфику конкретных устройств. Так, на TBDR-GPU для производительности графа в целом лучше группировать барьеры воедино, сортируя при этом граф соответствующим образом, в то время как на дискретных GPU барьеры, как правило, слабее влияют на производительность, что может позволить выбрать иную сортировку графа, способствующую уменьшению $LOAD$. Параллельное исполнение графа также зависит от специфики устройства, количества ядер, количества модулей обработки списков команд на GPU, из чего можно полгать, что вообще говоря, для различных устройств может быть оптимальным различное параллельное расписание исполнения вершин.

Кадровые графы — техника, совсем недавно получившая распространение в области компьютерной графики, но в перспективе способная улучшить все аспекты разработки приложений реального времени: потребление памяти, скорость работы приложения, и даже удобство разработки. Автор считает, что дальнейшие исследования, а также тяжкий труд по реализации теоретических идей на практике, позволят раскрыть этот потенциал и однажды сделать разработку приложений реального времени простым и приятным занятием.

Список литературы

1. *Valve, LLC.* Обзор технических характеристик персональных компьютеров пользователей «Steam». — URL: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam.
2. *Robson J. M.* An estimate of the store size necessary for dynamic storage allocation // Journal of the ACM (JACM). — 1971. — Т. 18, № 3. — С. 416—423.
3. *Advanced Micro Devices, Inc.* Драйвер памяти GPU, «VMA». — URL: https://gpuopen.com/vulkan-memory-allocator/.
4. *O'Donnell Y.* FrameGraph: Extensible Rendering Architecture in Frostbite. — 2017. — URL: https://www.gdcvault.com/play/1024612 ; Game Developers Conference.
5. *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.
6. *The Khronos® Group Inc.* Vulkan API Specification. — URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html.
7. *Wihlidal G.* Halcyon: Rapid innovation using modern graphics. — 2019. — URL: https://www.youtube.com/watch?v=da_6dsWz8yg ; Reboot Develop.
8. *Epic Games*. Unreal Engine rebder dependency graph. — URL: https://docs.unrealengine.com/5.0/en-US/render-dependency-graph-in-unreal-engine/.
9. *Tatarchuk N.*, *Aaltonen S.*, *Cooper T.* Unity Rendering Architecture. — 2021. — URL: https://www.youtube.com/watch?v=6LzcXPIWUbc ; SIGGRAPH 2021 REAC.

10. *Gruen H.* DirectX™ 12 Case Studies. — 2017. — URL: https://www.gdcvault.com/play/1024343 ; Game Developers Conference.
11. *Rodrigues T.* Moving to DirectX 12: Lessons Learned. — 2017. — URL: https://www.gdcvault.com/play/1024656 ; Game Developers Conference.
12. *Advanced Micro Devices, Inc.* Исходный код Render Pipeline Shaders. — URL: https://github.com/GPUOpen-LibrariesAndSDKs/RenderPipelineShaders.
13. *Advanced Micro Devices, Inc.* Анонс публикации Render Pipeline Shaders. — URL: https://gpuopen.com/learn/rps_1_0.
14. *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/.
15. *Stockmeyer I. J.* — 1976. — личная переписка.
16. *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.
17. *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.
18. *Kierstead H. A.* A polynomial time approximation algorithm for dynamic storage allocation // Discrete Mathematics. — 1991. — Т. 88, № 2. — С. 231—237. — Publisher: Elsevier.
19. *Gergov J.* Approximation algorithms for dynamic storage allocation // European Symposium on Algorithms. — Springer, 1996. — С. 52—61.

20. *Gergov J.* Algorithms for compile-time memory optimization // Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. — 1999. — С. 907—908.
21. 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.
22. *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.
23. *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.
24. *Yang L.*, *Liu S.*, *Salvi M.* A survey of temporal antialiasing techniques // Computer graphics forum. Т. 39. — Wiley Online Library. 2020. — С. 607—621.
25. *Barron K. B. D.* Super-sampling Anti-aliasing Analyzed. — 2000. — URL: http://www.x86-secret.com/articles/divers/v5-6000/datasheets/FSAA.pdf.
26. The Complexity of Coloring Circular Arcs and Chords / M. R. Garey [и др.] // SIAM Journal on Algebraic Discrete Methods. — 1980. — Т. 1, № 2. — С. 216—227. — DOI: 10.1137/0601025. — eprint: https://doi.org/10.1137/0601025. — URL: https://doi.org/10.1137/0601025.