## СОДЕРЖАНИЕ

[​ ВВЕДЕНИЕ 4](#__RefHeading___Toc1480_776179660)

[​ 1. МАТЕМАТИЧЕСКАЯ МОДЕЛЬ ГОЛОГРАММЫ 5](#__RefHeading___Toc89_2905274384)

[​ 2. ИСПОЛЬЗОВАНИЕ ГРАФИЧЕСКОГО ПРОЦЕССОРА ДЛЯ ПРОВЕДЕНИЯ МАТЕМАТИЧЕСКИХ РАСЧЕТОВ 6](#__RefHeading___Toc1484_776179660)

[​ 2.1. Устройство и принцип работы графического ускорителя 6](#__RefHeading___Toc309_2905274384)

[​ 2.2. Возникновение и развитие технологии GPGPU 13](#__RefHeading___Toc311_2905274384)

[3. ОБЗОР ФИЗИЧЕСКИХ УСТРОЙСТВ И ИСПОЛЬЗУЕМЫХ ПРОГРАММНЫХ ТЕХНОЛОГИЙ 15](#__RefHeading___Toc317_2905274384)

[​ 3.1. Обзор используемых аппаратных и программных средств 15](#__RefHeading___Toc319_2905274384)

[​ 3.2. Описание библиотеки MFC 15](#__RefHeading___Toc323_2905274384)

[​ 3.2.1. Общая структура программы на базе библиотеке MFC 16](#__RefHeading___Toc1608_57390669)

[​ 3.2.2. Иерархия классов библиотеки MFC 18](#__RefHeading___Toc3206_673334341)

[​ 3.2.2.1. Класс CObject 18](#__RefHeading___Toc3208_673334341)

[​ 3.2.2.2 Классы окон 19](#__RefHeading___Toc3210_673334341)

[​ 3.2.3. Карта сообщений 21](#__RefHeading___Toc3330_673334341)

[​ 3.3. Обзор спецификации OpenGL 21](#__RefHeading___Toc3212_673334341)

[​ 3.4. Обзор фреймворка OpenCL 21](#__RefHeading___Toc327_2905274384)

[​ 3.4.1 Модель платформы 22](#__RefHeading___Toc662_2926378771)

[​ 3.4.2 Модель исполнения 26](#__RefHeading___Toc922_2572051884)

[​ 3.4.3. Модель памяти 36](#__RefHeading___Toc924_2572051884)

[​ 3.4.4. Модель программирования 36](#__RefHeading___Toc1248_1363218577)

[​ 3.5. Совместное использование OpenGL и OpenCL 37](#__RefHeading___Toc534_2905274384)

[​ 3.5.1. Создание контекста OpenCL 38](#__RefHeading___Toc1492_1932796669)

[​ 3.5.2. Обмен данными между OpenCL и OpenGL 39](#__RefHeading___Toc1494_1932796669)

[​ 3.5.3. Синхронизация доступа к общим данным 40](#__RefHeading___Toc1496_1932796669)

[​ 4. РАЗРАБОТКА ПРОГРАММЫ 42](#__RefHeading___Toc1488_776179660)

[​ 4.1. Описание основного функционала программы 42](#__RefHeading___Toc435_2905274384)

[​ 4.2. Разработка функционала для инициализации и управления графическим ускорителем 43](#__RefHeading___Toc437_2905274384)

[​ 4.3. Разработка пользовательского интерфейса 46](#__RefHeading___Toc439_2905274384)

[​ 4.4. Разработка программ для графического процессора 46](#__RefHeading___Toc441_2905274384)

[​ 5. ТЕСТИРОВАНИЕ 47](#__RefHeading___Toc1490_776179660)

[​ ЗАКЛЮЧЕНИЕ 48](#__RefHeading___Toc1492_776179660)

[​ БИБЛИОГРАФИЧЕСКИЙ СПИСОК 49](#__RefHeading___Toc1494_776179660)

# ВВЕДЕНИЕ

# 1. МАТЕМАТИЧЕСКАЯ МОДЕЛЬ ГОЛОГРАММЫ

# 2. ИСПОЛЬЗОВАНИЕ ГРАФИЧЕСКОГО ПРОЦЕССОРА ДЛЯ ПРОВЕДЕНИЯ МАТЕМАТИЧЕСКИХ РАСЧЕТОВ

Большинство расчетов выполняются но центральном процессоре (ЦП). Основной акцент при производстве процессора ставится на времени выполнения инструкции, которое складывается из количества тактов, за которое выполняется инструкция. Чем меньше длительность одного такта, тем быстрее процессор выполняет инструкции. Но при использовании процессора необходимо учитывать ограничение по объему памяти и скорости его обработки процессором. (описать проблему). Данное ограничение можно компенсировать использованием графического процессора для проведения математических расчетов. В данной главе будет приведено описание использование графического процессора и технологии general-purpose computing on graphics processing units(GPGPU, ).

## 2.1. Устройство и принцип работы графического ускорителя

Конструкция графического процессора напоминает обычное вычислительное устройство, работающее отдельно от центрального процессора, параллельно с ним. Зачастую графические процессоры размещаются на отдельной плате, которая называется видеокартой или графическим ускорителем. Видеокарта имеет свою систему охлаждения и специализированную оперативную память — видеопамять, в которой хранятся обрабатываемые данные.

Основное назначение графического ускорителя заключается в параллельной обработки больших массивов данных. В связи с чем, архитектурное устройство всех графических процессоров имеет общие черты. Главным является система из параллельных вычислительных устройств, называемые ядрами (kernel) к различным элементам входных массивов данных, расположенных в общей памяти. Такая архитектура нацелена на исполнение алгоритмов, в которых элементы больших массивов данных обрабатываются одинаково почти или вовсе независимо друг от друга, то есть используется распараллеливание вычислений по данным. Элементы, которые выполняют однотипную независимую обработку называют потоками данных или результатов, так что графические процессоры осуществляют поточно-параллельную обработку данных.

Концепция программирования, ориентированная на потоковую обработку данных называется Single Instruction - Multiple Data (одна инструкция для множества данных) или сокращенно SIMD. Процессор, работающий по принципу SIMD, преобразует данные или поток данных в поток результатов, используя программу как функцию преобразования.

Выбор данной концепции программирования для графических процессоров обусловлена возможностью использования большого количества ядер без распределения задач, синхронизации и коммуникации. Это позволяет добить большей производительности и сокращает время программирования.

Помимо параллельной обработки данных присутствует и другая архитектурная особенность видеокарт, обусловленная специфичными потребностями компьютерной графики. Структура проведения операция на всех современных графический процессорах может быть представлена конвейером, устроенный таким образом, чтобы обеспечить максимальную эффективность выполнения задач компьютерной графики в условиях параллельной архитектуры. Суть заключается в том, что каждое из параллельных ядер может одновременно обрабатывать несколько элементов потока, применяя к ним различные операции. При этом, необходимо чтобы операции исполнялись одновременно и не зависели друг от друга.

Рассмотрим подробнее работу конвейера инструкций, ниже на рисунке (указать номер!!!) представлена графическая иллюстрация принципа конвейерной обработки данных.

|  |
| --- |
| Рис. . Этапы графического конвейера. |

Первый этап. Здесь определяется состояние объектов, принимающих участие в сцене, которую необходимо отобразить. На первый взгляд к самой графике этот этап отношения не имеет. На самом деле он является определяющим, ибо состояние объектов и их взаимное положение формируют логику последующих действий программы. Практически все операции на первом этапе выполняет центральный процессор. Результаты его работы пересылаются в графический чипсет посредством драйвера.

Второй этап. Происходит декомпозиция (разделение на примитивы) геометрических моделей. Внешний вид объекта формируется с помощью набора определенных примитивов. Чаще всего в роли примитива выступает треугольник как простейшая плоская фигура, однозначно располагаемая в трехмерном пространстве. Все прочие элементы состоят из таких треугольников. Таким образом, можно утверждать, что по большей части термины «полигон» и «треугольник» применительно к игровой SD-графике суть синонимы.

Современные графические процессоры умеют выполнять дополнительные операции, например тесселяцию (Tesselation), то есть разделение исходных треугольников на более мелкие. Некоторые графические чипсеты могут аппаратно обрабатывать геометрические модели, построенные на основе параметрических поверхностей (механизм RT-Patches). Часть графических процессоров умеет превращать плоские треугольники в трехмерные поверхности путем «выдавливания» в третье измерение (механизм N-Patches). Итоговый результат операций второго этапа пересылается в блок трансформаций и освещения (Transform&Lighting, T&L) геометрического процессора.

Третий этап. В блоке T&L на аппаратном уровне к вершинам треугольников применяют различные эффекты преобразований и освещенности. Содержание операций блоков T&L новейших графических чипсетов можно динамически изменять посредством вершинных шейдеров (Vertex Shaders) — специальных микропрограмм, включаемых в код игры. То есть сегодня персональный компьютер в дополнение к центральному процессору получил полноценный программируемый графический процессор. По завершении операций трансформации и расчета освещенности параметры вершин нормализуются и приводятся к целочисленному виду.

Четвертый этап. На данном этапе происходит так называемая установка примитивов (Triangle Setup). Графический процессор пока ничего не знает о свойствах треугольников, поскольку обрабатывал вершины по отдельности. Теперь необходимо «собрать» вершины в треугольники и преобразовать результаты в координаты и цвет каждого пиксела, а также отсечь невидимые области.

В ходе «сборки» определяется видимость объектов с позиции камеры. Полигоны, находящиеся ближе к камере, могут загородить более удаленные полигоны. Для хранения информации о степени удаленности объекта от плоскости проецирования используют специальный буфер глубины (Z-буфер). Современные графические процессоры применяют различные механизмы отсечения невидимых полигонов на ранних этапах ЗD-конвейера с тем, чтобы избежать излишних операций. Данные буфера глубины обрабатываются специализированными блоками графического процессора. В конечном счете, на выходе блока геометрических преобразований получают проекцию трехмерной сцены на плоскость визуализации. Координаты и исходный цвет видимых пикселов передаются в текстурный конвейер (Texture Pipeline).

Пятый этап. Графический чипсет может иметь несколько параллельных текстурных конвейеров. В каждом из них происходит наложение текстур различного типа, в том числе и тех, которые сами не отображаются (например, карты высот), а служат для модификации других текстур. На этом этапе в современных чипсетах возможно исполнение пиксельных шейдеров (Pixel Shaders) — специальных микропрограмм, определяющих порядок смешивания текстур, полученных на выходе из каждого конвейера. Здесь также учитывается информация (полученная на этапе установки примитивов) о принадлежности пиксела к определенному треугольнику. В целом указанные операции составляют суть процесса визуализации (рендеринга).

Шестой этап. На заключительном этапе работы ЗD-конвейера к полученному в текстурном блоке плоскому изображению применяют операции устранения дефектов (например, часто используют билинейные, анизотропные и другие фильтры). Операции конечной обработки применяют какой-либо эффект (например, туман) к целиком сформированному изображению. Перед выводом в буфер кадра в последний раз проверяется видимость пикселов, и наконец изображение появляется на экране.

Перечисленные этапы в конкретных графических чипсетах могут быть переставлены, разделены, объединены, выполняться неоднократно (в несколько проходов), однако их физический смысл остается неизменным. Технологически некоторые элементы этапов или этапы целиком могут быть выполнены различными способами. Вариант реализации зависит от особенностей приложения и видеокарты.

В зависимости от типа видеоускорителя часть этапов просчитывается программно, а часть — аппаратно. Самые современные ускорители имеют на борту графический процессор, способный аппаратно просчитывать этапы трансформации (преобразований) и расчета освещения, наложения текстур.

Графический процессор управляется пользователем не напрямую, а через центральный процессор компьютера. При этом CPU в большинстве случаев исполняет как минимум три взаимодействующие программы, которые различаются по происхождению, назначению и возможностям. Эти программы работают на трех уровнях управления графическим процессором. На двух уровнях, «ближайших» к графическому процессору, исполняются стандартные программы, избавляющие программиста от необходимости написания рутинных процедур, таких, например, как выделение видеопамяти для данных или представление этих данных в формате, необходимом графическому процессору.

На самом низком уровне, «ближайшем» к графическому процессору, исполняется драйвер графического процессора — программа, которая непосредственно управляет как самим GPU, так и видеопамятью. Фактически драйвер работает как часть операционной системы. Он получает от других приложений запросы — задачи для графического процессора, после чего передает их на GPU в оптимальном порядке и необходимом формате. В частности, функциями драйвера являются такие:

* распределение ресурсов графического процессора между

несколькими приложениями;

* загрузка данных в видеопамять и регистры графического про‑

цессора, копирование данных из видеопамяти в оперативную

память компьютера;

* загрузка в графический процессор и запуск на исполнение поль‑

зовательских программ;

* автоматическое распределение расчетов между параллельными

вычислительными блоками графического процессора;

исполнение стандартных алгоритмов обработки графики;

* регулирование частоты графического процессора для поддержания рабочей температуры.

Драйверы обычно разрабатываются самими производителями графических процессоров, поскольку это требует детального знания их технического устройства и особенностей. Они обновляются для поддержки процессоров новых моделей. Благодаря существованию драйверов, пользовательские программы могут выполняться на различных GPU без перекомпиляции, если GPU и драйвер поддерживают функции, задействованные в программе.

К драйверу GPU можно обращаться из пользовательских приложений напрямую, однако это требует явного включения в текст программы большого количества стандартных операций управления графическим процессором. Для того чтобы автоматизировать выполнение подобных операций и унифицировать взаимодействие пользовательских приложений с различными устройствами различных производителей, существуют библиотеки специализированных процедур, известные как программный интерфейс приложений (application programming interface, или API). В качестве разработчиков API для графических процессоров выступают как сами производители GPU, так и компании, специализирующиеся на программном обеспечении. Под управлением Windows наиболее широко используется пакет API DirectX от Microsoft, а для других операционных систем такие библиотеки разрабатываются в основном по спецификациям OpenСL и OpenGL. Подробнее эти интерфейс буду рассмотрены в главе 3.

На «верхнем» уровне управления графическим процессором находятся пользовательские приложения — программы, разрабатываемые для решения конечных задач, таких как создание трехмерных сцен, проведение физико‑математических расчетов. Структура пользова тельского приложения, использующего API типа DirectCompute или OpenCL, показана на рис. 1.6.

При использовании DirectCompute или OpenCL пользовательское приложение состоит по крайней мере из двух частей, для написания которых используются различные языки программирования. Одна из этих частей — это программа для графического процессора, которую обычно называют вычислительным ядром, а другая часть — центральный процессор .Для написания вычислительных ядер используются специальные языки программирования. При работе с DirectX это может быть либо язык высоко уровня HLSL, либо низкоуровневый ассемблер (assembly shader language). Вычислительное ядро хранится в отдельном текстовом файле, оно компилируется и загружается в память GPU средствами DirectX или OpenCL непосредственно перед запуском. Эти API используются также для загрузки в графический процессор и видеопамять данных и констант, необходимых для расчета.

## 2.2. Возникновение и развитие технологии GPGPU

Основным критерием выбора того или иного вычислительного устройства является его производительность. Этот термин означает количественную характеристику скорости выполнения определенных операций на вычислительном устройстве. Чаще всего производительность определяется на вещественными данными в секунду (floating-point operations per seconds, FLOPS). Также определяют пиковую и реальную производительность. Первая используется при учете только скорости расчета, а вторая при влиянии других факторов, таких как память.

Производительность сильно зависит от тактовых частот центрального процессора и памяти. Так, процессоры с высокой тактовой частотой и большим объем интегрированной памяти могли бы обладать хорошей производительность, но из-за сложности производства подобных чипов их цена будет очень большой. По этой причине основной объем памяти находится отдельно от центрального процессора в внешних модулях и частота их работы во много раз ниже частоты центрального процессора. Таким образом реальная производительность системы обработки данных, в много раз меньше пиковой производительности системы.

История развития графического процессора насчитывает большое количество различных этапов. Как минимум промежуточным итогом для данного времени можно считать то, что графический ускоритель эволюционировал от простого устройства для растеризации графики, до полноценного вычислительного устройства с производительность порядка Терафлопс и собственным API.

Универсальные вычисления на графических процессорах (GPGPU) — это использование графического процессора (GPU), который обычно выполняет только для компьютерной графики, для выполнения вычислений в приложениях, которые традиционно обрабатываются центральный процессор (CPU). Использование нескольких видеокарт в одном компьютере или большого количества графических чипов дополнительно распараллеливает и без того параллельный характер обработки графики. Кроме того, даже одна структура GPU-CPU обеспечивает преимущества, которые несколько процессоров сами по себе не могут предложить из-за специализации каждого чипа.

По сути, GPGPU pipeline является своего рода параллельная обработка между одним или несколькими GPU и CPU, которая анализирует данные, как если бы они были в виде изображения или другой графической формы. Хотя графические процессоры работают на более низких частотах, у них обычно во много раз больше ядер . Таким образом, графические процессоры могут обрабатывать гораздо больше изображений и графических данных в секунду, чем традиционные процессоры. Перенос данных в графическую форму с последующим использованием графического процессора для их сканирования и анализа может создать большое ускорение.

Конвейеры GPGPU были разработаны в начале 21 века для обработки графики (например, для лучше шейдеры ). Было обнаружено, что эти конвейеры хорошо удовлетворяют потребности научных вычислений , и с тех пор были разработаны в этом направлении.

В принципе, любая произвольная логическая функция , включая функции сложения, умножение и другие математические функции могут быть построены из функционально полного набора логических операторов. В 1987 году Game of Life Конвея стала одним из первых примеров вычислений общего назначения с использованием раннего потокового процессора , называемого блиттером , для вызова специальной последовательности логические операции с битовыми векторами.

Вычисления общего назначения на графических процессорах стали более практичными и популярными примерно после 2001 года, с появлением как программируемых шейдеров , так и плавающих Поддержка пункта на графических процессорах. Примечательно, что проблемы, связанные с матрицей и / или векторами , особенно двух-, трех- или четырехмерными векторами, было легко преобразовать в графический процессор, который работает с собственной скоростью и поддержкой. по этим типам. Эксперименты научного компьютерного сообщества с новым оборудованием начались с процедуры матричного умножения (2001); одной из первых общепринятых научных программ, которые работали на GPU быстрее, чем на CPU, была реализация LU-факторизации (2005).

Эти ранние попытки использовать GPU в качестве процессоров общего назначения потребовали переформулирования вычислительных проблемы с графическими примитивами, которые поддерживаются двумя основными API-интерфейсами для графических процессоров, OpenGL и DirectX . Этот громоздкий перевод был устранен с появлением языков программирования общего назначения и API, таких как Sh/RapidMind , Brook и Accelerator.

За ними последовала Nvidia CUDA , который позволял программистам игнорировать лежащие в основе графические концепции в пользу более общих концепций высокопроизводительных вычислений . Новые предложения, не зависящие от производителей оборудования, включают DirectCompute от Microsoft и OpenCL от Apple / Khronos Group. Это означает, что современные конвейеры GPGPU могут использовать скорость графического процессора, не требуя полного и явного преобразования данных в графическую форму.

# 3. ОБЗОР ФИЗИЧЕСКИХ УСТРОЙСТВ И ИСПОЛЬЗУЕМЫХ ПРОГРАММНЫХ ТЕХНОЛОГИЙ

## 3.1. Обзор используемых аппаратных и программных средств

## 3.2. Описание библиотеки MFC

Разработка программного обеспечения по операционную систему (ОС) Windows, любой версии, не обходится без использования фреймворка WinAPI. Эта технология позволяет программистам создавать графический интерфейс для своих программ и использовать прочие ресурсы ОС Windows, такие как управление процессами и потоками, систему ввода-вывода и так далее. Но писать программы с использованием WinAPI непростая задача. Необходимо иметь большой опыт в использовании данного фреймворка, чтобы процесс разработки программы не отнимал большого количества времени и сил. В связи с этим, были созданы дополнительные библиотеки для использования WinAPI в более простой и понятной парадигме объектно-ориентированного программирования. Одной из таких библиотек является MFC (Microsoft Foundation Classes).

Библиотека MFC является инструментом, который берет на себя большую часть черной работы. MFC включает в себя большое количество классов, каждый из которых предоставляет определенный функционал для работы с ОС Windows, а также предоставляет новые механизмы, которые упрощаю разработку программного обеспечения для ОС Windows.

Перед создателями библиотеки стояла задача создания объектно-ориентированного интерфейса для Windows, удовлетворяющего следующим основным целям проектирования программных продуктов:

* сокращение усилий по программированию приложений для Winodws;
* скорость выполнения программ, написанных с использованием библиотеки, должна быть сопоставима с программами, написанными на языке C с использованием Win32 API;
* минимальный размер вспомогательного кода;
* способность напрямую вызывать любую C-функцию Win32 API;
* легкость использования Win32 API в C++ должна быть такая же, как и при использовании традиционного C.

Поставленная задача была выполнена разработчиками библиотеки на высоком уровне. Созданная библиотека классов охватывает все компоненты Windows – окна, блоки диалога, контексты устройств, общие объекты GDI (битовые образы и кисти), элементы управления и многие другие стандартные элементы. Основной парадигмой разработки ПО под операционную систему Windows является обработка событий. При использовании Win32 API программист должен самостоятельно позаботиться об обработке сообщений от того или иного элемента управления. В следствии этого программный код становится громоздким из-за большого количества операторов switch или условных операторов if … else if … else. Этот недостаток был решен разработчиками библиотеки MFC. Классы библиотеки полностью вобрали в себя многочисленные условные операторы языка программирования, которые так загромождают программный код на языке C/C++.

В данном разделе будет рассмотрена лишь общая структура построения программ на основе библиотеки MFC. Будут затронуты такие темы как иерархия классов, архитектура документ-представление и карта сообщений. Для более подробного ознакомления с устройством библиотеки следует обратиться к (указать!!!).

## 3.**2**.1. Общая структура программы на базе библиотеке MFC

Как уже было сказано ранее библиотека MFC предоставляет набор классов для взаимодействия с операционной системой Windows. Основным отличием программы написанной на языке программирования C, является то, что точка входа (функция WinMain) не задается самим программистом явно, она скрыта в реализации библиотеке MFC. Отправной точкой программа, написанных с использованием библиотеке MFC является класс CwinApp. Его структура (поля и методы) зависит от конкретного приложения, но есть и общие поля и методы, речь о них пойдет дальше. Сейчас же сосредоточимся на назначении данного класса. Основной задачей этого класса является создание процесса программы в системе, инициализация и создание главного окна и опрос системы сообщений. Под процессом в ОС Windows понимают каждый выполняемый экземпляр приложения, то становится ясно, что в приложении может быть определен только один объект класса CWinApp. Зачастую программист создает собственный класс для запуска и своего приложения в системе, для этого ему необходимо наследоваться от класса CWinApp и дополнить его на свое усмотрение. Пример подобного класса будет рассмотрен в дальнейшем при обсуждении процесса разработки приложения.

Разработчики библиотеки проделали большую работу и избавили программиста от инициализации и определения функции WinMain и многих других строк вспомогательного кода. Класс CWinApp выполняет все действия, которые обычно выполняла функция WinMain - инициализирует приложение, обрабатывает сообщения и завершает приложение. Для этого класс CWinApp включает виртуальные методы InitApplication XE "CWinApp:InitApplication" , InitInstance XE "CWinApp:InitInstance" , Run XE "CWinApp:Run" и ExitInstance XE "CWinApp:ExitInstance" .

Чтобы выполнить инициализацию приложения, функция WinMain вызывает методы InitApplication и InitInstance для объекта главного класса приложения. Метод InitApplication выполняет инициализацию на уровне приложения. Вы можете переопределить этот метод в своем приложении. Метод InitInstance выполняет инициализацию каждой запущенной копии приложения. Обычно метод InitInstance создает главное окно приложения. Вы обязательно должны переопределить этот метод в своем приложении. Остальные методы, например Run, можно не переопределять.

Затем функция WinMain начинает обрабатывать цикл сообщений. Для этого вызывается метод Run. Вы можете переопределить этот метод, чтобы реализовать собственный цикл обработки сообщений.

Когда приложение заканчивает работу и цикл обработки сообщений завершается, вызывается метод ExitInstance. Вы можете переопределить этот метод, чтобы выполнить какие-либо действия перед завершением приложения.

## 3.**2**.2. Иерархия классов библиотеки MFC

Основным достоинством библиотеки MFC является ее объектно-ориентированная архитектура. Библиотека предоставляет разработчику большое количество классов для разработки программного обеспечения для операционной системы Windows. Описание всех имеющихся классов будет слишком громоздким и излишним, поэтому ограничимся только рассмотрением базовых и основополагающих вопросов архитектуры библиотеки MFC. Другими словами в этом разделе рассмотрим лишь те классы, которые необходимы для написания программ и необходимых для реализации данного проекта.

## 3.**2**.2.1. Класс CObject

Все классы библиотеке объединены в иерархию, представленную на рис (указать номер!!!). На вершине иерархии располагается класс CObject. Основное назначение этого класса заключается в предоставлении всем своим производным классам следующие возможности: хранение информации о классе объекта во время выполнения и поддержка сериализации и диагностики объекта.

Класс CObject хранит в себе единственную статическую переменную classObject типа CRuntimeClass, которая хранит в себе информацию об объекте, наследуемого от класса CObject, во время выполнения программы. Так же класс CObjetc не поддерживает множественного наследования. Данный класс предоставляет четыре уровня функциональных возможностей:

1. Базовые функциональные возможности – отсутствует поддержка динамической информации о классе и сериализации, но включает поддержка диагностики памяти.
2. Базовые функциональные возможности плюс поддержка динамической информации о классе.
3. Базовые функциональная возможности плюс поддержка динамической информации о классе и динамическое создание.
4. Базовые функциональные возможности плюс поддержка динамической информации о классе, динамическое создание и сериализация.

Динамическую информацию о классе CObject поддерживает посредством определенного метода IsKindOf, который позволяет определить, относится ли объект к определенному классу или является производным от него. Эта функция позволяет осуществить безопасное приведение типа в производном классе. Ее аргументом является объект класса CRuntimeClass, который можно получить, используя макрос RUNTIME\_CLASS с именем класса. Метод IsKindOf не поддерживает множественного наследования или виртуальных классов, хотя при необходимости можно использовать множественное наследование для классов, производных от классов библиотеке MFC.

После рассмотрения корневого класса иерархии MFC перейдем к обсуждению остальных классов. Остальные классы библиотеке можно разбить на две категории: производные от класса CObject и классы, не зависящие от него. Для наших целей будут интересны только некоторые классы данной иерархии, за подробным рассмотрением остальных классов можно обратиться к (указать источник!!!).

## 3.**2**.2.2 Классы окон

Вторым по значимости классом является класс CWnd – базовый класс для всех окон, созданных на базе библиотеки MFC. Назначение этого класса состоит в предоставлении интерфейса всем другим оконным классам библиотеки MFC и создание дочерних окон Windows. Класс CWnd предоставляет большое количество разнообразных функций для разработчика программного обеспечения. Для простоты изложения приведем список категорий с краткой характеристикой входящих в него методов:

* Конструктор – создает объект класса без создания окна Windows.
* Инициализация – отвечает за создание окна Windows (соответствующей ему внутренней структуры) и присоединение его к объекту класса. В состав этой категории входят функции: Create (CreateEx), PreCreateWindow, GetStyle (GetExStyle), Attach, Detach, GetSafeWnd, FromHandel, FromPermanentHandle, PreSubclassWindow, SubclassWindow.
* Состояние окна – отвечает за определение (функция) или изменения (функция) доступа к окну; определение или назначение активного окна (функции); получить или установить фокус ввода с клавиатуры (функции) или от мыши (функции), а также дескриптом пиктограммы (функции); стиль окна (функции) и указатель на объект окна.
* Изменение размеров и позиции окна – функции, входящие в данную группу отвечают за определение или установку состояния окна и позволяющие узнать или изменить размеры и/или позицию окна. (Функции).
* Доступ к окну – функции данной группы предоставляют возможность определить существует ли в системе окно. (Функции).
* Обновление-перерисовка – отвечают за рисование внутри окна. (Функции).
* Преобразование координат – служит для взаимного преобразования рабочих и экранных координат текущего окна.
* Работа с текстом – в эту категорию входят функции, которые позволяют установить или изменить текст, ассоциированный с окном. (Функции).
* Работа с сообщениями – функции этой категории необходимы для перехвата и обработки сообщений операционной системы.

Остальные функции имеют достаточно специфичную направленность и служат больше для пользовательского интерфейса.

Дочерний класс CFrameWnd класса CWnd служит для создания перекрывающихся или всплывающих окон и поддерживает однодокуметный интерфейс Windows (SDI). С этим классом (и производным от него) тесно связано понятие фрейма (frame window). Под ним понимается окно, которое координирует взаимодействие пользователя с приложением. Оно отображается на экране в виде рамки с необязательной строкой состояния и стандартными элементами управления. Это окно отвечает за управление размещением своих дочерних окон и других элементов рабочей области. Кроме того, фрейм переадресует команды своим представлениям (специальным дочерним окнам) и может отвечать на сообщения от элементов управления.

## 3.**2**.3. Карта сообщений

## 3.**3**. Обзор спецификации OpenGL

## 3.**4**. Обзор фреймворка OpenCL

В официальной документации производителя стандарта Khronos сказано, что OpenCL (Open Computing Language) является открытым и бесплатным отраслевым стандартом параллельного программирования общего назначения для центрального процессора, графического процессора и других вычислительных устройств. OpenCL – это среда для параллельного программирования, включающая язык, API, библиотеки и систему времени выполнения для поддержки разработки программного обеспечения. Основным преимуществом данного стандарта является сего переносимость на разные платформы, чем заслужил большую популярность среди разработчиков, ориентированных на работу с гетерогенными системами.

Иерархия OpenCL включает в себя 4 уровня:

* модель платформы;
* модель исполнения;
* модель памяти;
* модель программирования.

Рассмотрим каждый из уровней подробнее.

## 3.**4**.1 Модель платформы

Под платформой принято понимать совокупность хоста и набора устройств, управляемых OpenCL, которые позволяют приложения обмениваться ресурсами и выполнять ядра на устройствах в платформе. В качестве хоста используется центральный процессор (ЦП, CPU) вычислительного устройства, а устройства OpenCL (устройства, devices) — некоторый набор вычислительных единиц, который соответствует графическому процессору (ГП, GPU), многоядерному центральному процессору или другим процессорам с параллельной архитектурой, доступных для управления из хост-программы.

На рисунке (номер!!!) определена модель платформы для OpenCL, предоставленная документацией разработчика стандарта. Модель состоит из хоста, подключенного к одному или нескольким устройствам OpenCL. Устройство OpenCL делится на один или несколько вычислительных блоков (CU), которые далее делятся на один или несколько элементов обработки (PE). Вычисления на устройствах происходят внутри обрабатывающих элементах.

|  |
| --- |
| Рис. . Модель платформы. |

Приложение OpenCL реализовано как в виде кода хоста, так и кода ядра устройства. Код хоста передает код ядра для устройства OpenCL в виде команд. Устройство OpenCL выполняет вычисление команд на обрабатывающих элементах внутри устройства.

Программы для устройств OpenCL представляются в виде исходных двоичных файлов SPIR-V, исходных строк OpenCL C или OpenCL C ++ или двоичных объектов, определенных реализацией. Платформа OpenCL предоставляет компилятор для преобразования ввода программы любой формы в исполняемые программные объекты.

Для платформ OpenCL существует два типа профилей: полный профиль и встроенный профиль. Полнопрофильная платформа должна предоставлять онлайн-компилятор для всех своих устройств. Встроенная платформа может предоставлять онлайн-компилятор, но это не обязательно.

Устройство может предоставлять специальные функциональные возможности как встроенную функцию . Платформа предоставляет API-интерфейсы для перечисления и вызова встроенных функций, предлагаемых устройством, но в остальном не определяет их конструкцию или семантику. А пользовательские устройства поддерживают только встроенные функции, и не может быть запрограммирован с помощью языка ядра.

Для определения платформ и устройств, а также информации об них используются функции OpenCL API. Для определения платформы используется функция clGetPlatformIDs:

Первый аргумент определяет количество платформ, которые будут переданы во второй аргумент, третий аргумент передает количество доступных платформ.

cl\_int clGetPlatformIDs(

cl\_uint num\_entries ,

cl\_platform\_id \*platforms ,

cl\_uint \*num\_platforms );

Рассмотрим вызов функции на примере, приведенном на листинге (номер!!!). В начале создается переменная platforms, которая является указателем на структуру cl\_platform\_id, изначально ей присваивается значение nullptr. Вторая переменная num\_platforms типа cl\_uint необходима для определения количества доступных OpenCL платформ. Следующим шагом вызываем функцию clGetPlatformIDs, первому аргументу передаем значение 0, во второй аргумент передаем nullptr, в третий аргумент передаем ссылку на переменную num\_platforms. После выполнения функции в переменной num\_platfroms будет записано количество OpenCL платформ, в переменную err передается код ошибки вызова функции. Затем проводим проверку значения переменной err и num\_platforms. Если переменная err не равна CL\_SUCCESS, что соответствует удачному вызову функции или значение в переменной num\_platforms осталось нулевым, то останавливает работу программы. Иначе выделим память под платформы OpenCL при помощи фукнции new, затем проведем повторный вызов функции clGetPlatformIDs. При повторном вызове функции в первый аргумент передадим значение переменной num\_platforms, во второй аргумент передадим указатель platforms, в третий аргумент передаем NULL. Затем вновь проверяем код ошибки вызова функции. В случае успешного вызова в переменной platforms будут храниться структуры cl\_platform\_id, в количестве определенном в переменной num\_platforms.

Для получения информации о платформе OpenCL используется функция clGetPlatformInfo():

cl\_platform\_id\* platfroms = nullptr;

cl\_uint num\_platfroms = 0;

cl\_int err = clGetPlatformIDs(0, platfroms, &num\_platfroms);

if (err != CL\_SUCCESS || num\_platfroms == 0)

exit(1);

platfroms = new cl\_platform\_id[num\_platfroms];

err = clGetPlatformIDs(num\_platfroms, platfroms, NULL);

if (err != CL\_SUCCESS)

exit(1);

Для заданной платформы platform указывается тип необходимой дополнительной информации prm\_name (с помощью заранее предопределённых констант CL\_PLATFORM\_PROFILE, CL\_PLATFORM\_VERSION, CL\_PLATFORM\_NAME, CL\_PLATFORM\_VENDOR, CL\_PLATFORM\_EXTENSIONS); остальные параметры используются в описанном выше стиле «двойного вызова».

cl\_int clGetPlatformInfo(

cl\_platform\_id platform ,

cl\_platform\_info prm\_name ,

size\_t prm\_value\_size ,

void \*prm\_value ,

size\_t \*prm\_value\_size\_ret );

После инициализации платформы OpenCL необходимо провести инициализацию устройств OpenCL. Делается это при помощи функции clGetDeviceIDs():

Данная функция во многом схожа с предыдущей функцией с тем отличием, что в первых двух аргументах необходимо передать указатель на необходимую платформу и тип инициализируемого OpenCL устройства. Назначение последних трех аргументов аналогично функции clGetPlatformIDs. Также и использование данной функции аналогично предыдущей. Вначале вызов функции необходим для определения количества доступных устройств, вторым вызовом происходим инициализация указателя devices.

cl\_int clGetDeviceIDs(

cl\_platform\_id platform ,

cl\_device\_type device\_type ,

cl\_uint num\_entries ,

cl\_device\_id \*devices ,

cl\_uint \*num\_devices );

Для получения информации об устройстве OpenCL используется функция clGetDeviceInfo():

Использование же данной функцией схоже с применением функции для получения информации о платформе.

cl\_int clGetDeviceInfo(

cl\_device\_id device ,

cl\_device\_info param\_name ,

size\_t prm\_value\_size ,

void \*prm\_value ,

size\_t \*param\_value\_size\_ret );

Таким образом, модель платформы - это абстракция, описывающая, как OpenCL рассматривает оборудование. Взаимосвязь между элементами модели платформы и аппаратным обеспечением в системе может быть фиксированным свойством устройства или может быть динамической функцией программы, зависящей от того, как компилятор оптимизирует код для наилучшего использования физического оборудования.

## 3.**4**.2 Модель исполнения

cl\_context clCreateContext (

const cl\_context\_properties \*properties ,

cl\_uint num\_devices ,

const cl\_device\_id \*devices ,

void (CL\_CALLBACK \*pfn\_notify )(

const char \*errinfo ,

const void \*private\_info , size\_t cb ,

void \*user\_data ),

void \*user\_data ,

cl\_int \*errcode\_ret );

cl\_context clCreateContextFromType (

const cl\_context\_properties \*properties ,

cl\_device\_type device\_type ,

void (CL\_CALLBACK \*pfn\_notify )(

const char \*errinfo ,

const void \*private\_info , size\_t cb ,

void \*user\_data ),

void \*user\_data ,

cl\_int \*errcode\_ret );

Выполнения вычислений на устройстве OpenCL происходит в несколько этапов:

1. создание контекста;
2. создание ядра, выделение памяти и передача аргументов ядра;
3. создание очереди команд;
4. отображение рабочих элементов и выполнение ядра.

Данный процесс программисту необходимо выполнять явно, для этого ему необходимы функции OpenCL API. Рассмотрим каждый этап и функции, необходимые для его прохождения, подробнее.

**На первом этапе** происходит создание контекста исполнения. Под контекстом понимают среду, в которой выполняются ядра, и домен, в котором определены синхронизация и управление памятью. Контекст включает в себя множество устройств , память , доступную для этих устройств , соответствующих свойств памяти и одного или несколько командных очередей , используемых для выполнения расписания в виде ядра (ов) или операции на объектах памяти. За создание контекста отвечают две функции clCreateContext() и clCreateContextFromType():

Отличие одной функции от другой заключается в том, что для первой необходимо передавать структуру cl\_device\_id, созданную заранее, а для второй достаточно указать тип необходимо устройсва. В остальном они принимают совершенно одинаковые параметры: указатель properties на массив свойств контекста, указатель pfn\_notify на функцию обратного вызова (со своим списком параметров), указатель user\_data на пользовательские данные, если они будут необходимы для функции обратного вызова. Возможные ошибочные ситуации при вызове любой из функций создания контекста фиксируются в переменной типа cl\_int, указатель на которую передаётся через последний параметр errcode\_ret .

Рассмотрим небольшой пример создания контекста, приведенный на листинге (номер!!!).

В данном примере происходит вызов функции clCreateContext(). В качестве свойств контекста передается значение NULL. Второй и третий параметры определяют количество и сами устройства, инициализированные заранее, в данном примере создается контекст для одного устройства. В четвертый и пятый элемент передаются значения NULL. В последний аргумент передается ссылка на переменную, отвечающая за хранение кода ошибки.

cl\_int err = CL\_SUCCESS;

cl\_context\_properties \*prop = NULL;

cl\_context m\_Context = clCreateContext(prop, 1, &dev, NULL, NULL, &err);

if (err != CL\_SUCCESS)

exit(1);

**Второй этап** предполагает создания ядра, выделение объектов памяти и передача этих объектов ядру. Ядро — это объект, который выполняется на одном или нескольких устройствах OpenCL в четко определенном контексте, управляемый хостом.

Для создания ядра необходимо проделать следующую процедуру. В начале необходимо написать программу, которая будет выполняться на устройстве (-ах) OpenCL. Подробнее этот момент будет рассмотрен в одном из следующих пунктов(указать номер!!!), сейчас же ограничимся тем, что программа уже создана и находится в файле с расширением «.cl». Теперь необходимо провести загрузку и компиляцию этой программы. Для этого в OpenCL API предусмотрен особый тип данных cl\_program, его созданием занимается функция clCreateProgramWithSource():

Здесь context — контекст, strings — массив указателей на символьные строки (в количестве count ), которые могут быть как завершены нулевым символом, так и не использовать его; если «завершителя» у строк нет, их длина должна быть указана в массиве длин lengths . Возможные ошибочные ситуации при вызове фиксируются в переменной типа cl\_int, указатель на которую передаётся через последний параметр errcode\_ret . Результатом вызова будет программный объект типа cl\_program.

cl\_program clCreateProgramWithSource (

cl\_context context ,

cl\_uint count ,

const char \*\*strings ,

const size\_t \*lengths ,

cl\_int \*errcode\_ret );

После создания объекта программы, необходимо в провести ее компиляцию, делается это функцией clBuildProgram():

Для построения исполняемого кода ядер необходимо указать программный объект prog , полученный на предыдущем шаге (здесь различаются и по-разному выделены имя фиктивного параметра prog в прототипе функции и реально используемая при вызове величина prg); задать — если это необходимо — список устройств device\_list , для которых должно быть осуществлено построение, и количество этих устройств num\_devices (иначе производится построение для всех устройств, связанных с программой); перечислить дополнительные опции options компиляции и линковки (фактически — параметры командной строки компилятора и линкера); задать — если требуется — адрес нотифицирующей функции, которая будет вызвана после завершения построения с передаваемыми ей значениями prog и udata в качестве необходимых параметров.

cl\_int clBuildProgram (

cl\_program prog ,

cl\_uint num\_devices ,

const cl\_device\_id \*device\_list ,

const char \*options ,

void (CL\_CALLBACK \*pfn\_notify )(cl\_program prog , void \*udata ),

void \*udata );

В случае возникновения каких-либо ошибок (возвращаемых здесь в переменную с именем err) OpenCL API предусмотрена возможность вывода подробной информации об ошибке компиляции программы, отвечает за это функция clGetProgramBuildInfo() со специальным флагом CL\_PROGRAM\_BUILD\_LOG:

Нужный тип информации для программного объекта program и устройства device задаётся значением параметра prm\_name (CL\_PROGRAM\_BUILD\_STATUS, CL\_PROGRAM\_BUILD\_OPTIONS или CL\_PROGRAM\_BUILD\_LOG), prm\_value — указатель на участок памяти, куда возвращается информация, prm\_value\_size — размер информации в байтах, prm\_value\_size\_ret — указатель на переменную, в которую помещается размер информации в байтах при запросе, когда prm\_value равен NULL; если размер уже известен, параметр prm\_value\_size\_ret может иметь значение NULL. Лучше всего вызывать функцию дважды: сначала для определения размеров возвращаемого лога, а затем — после выделения для него необходимой памяти — для получения собственно лога.

cl\_int clGetProgramBuildInfo (

cl\_program program ,

cl\_device\_id device ,

cl\_program\_build\_info prm\_name ,

size\_t prm\_value\_size ,

void \*prm\_value ,

size\_t \*prm\_value\_size\_ret );

Для большей наглядности рассмотрим пример создания и клмпиляции программы OpenCL, приведенном на листинге (указать номер!!!).

В начале происходит создание объекта STL библиотеки для взаимодействия с системой ввода-вывода и создание объекта класса std::string, тоже определенного в библиотеке STL. На следующем шаге происходит создание объекта cl\_program функцией clCreateProgramWidthSource, описанной ранее. Затем производится компиляция программы и вывод сообщения об ошибке компиляции в случае неудачи.

std::ifstream file(m\_ProgramPath, std::ios\_base::binary);

std::string code(std::istreambuf\_iterator<char>(file), (std::istreambuf\_iterator<char>()));

const char\* src = code.c\_str();

std::size\_t len = code.length() + 1;

m\_Program = clCreateProgramWithSource(m\_Context, 1, &src, &len, &err);

if (err != CL\_SUCCESS)

exit(1);

cl\_int res = clBuildProgram(m\_Program, 0, NULL, NULL, NULL, NULL);

if (res != CL\_SUCCESS)

{

std::size\_t size\_log = 0;

char\* build\_log;

res = clGetProgramBuildInfo(m\_Program, dev, CL\_PROGRAM\_BUILD\_LOG, 0, nullptr, &size\_log);

if (res != CL\_SUCCESS)

exit(1);

if (size\_log == 0)

exit(1);

build\_log = new char[size\_log];

res = clGetProgramBuildInfo(m\_Program, dev, CL\_PROGRAM\_BUILD\_LOG, size\_log, build\_log, nullptr);

if (res != CL\_SUCCESS)

exit(1);

std::cout << build\_log << std::endl;

delete[] build\_log;

}

Заключительным шагом процедуры создания ядра является создание объекта cl\_kernel, создание объектов памяти cl\_mem и передача их ядру в качестве аргументов функции. Для создания объекта ядра необходима программа и название самой функции ядра, отвечает за это функция clCreateKernel():

Здесь program — програмный объект с успешно построенным исполняемым кодом ядра, kernel\_name — имя функции со спецификатором \_\_kernel, errcode\_ret — указатель на переменную для кода ошибки. В результате возвращается объект ядра, готовый для запуска.

cl\_kernel clCreateKernel (

cl\_program program ,

const char \*kernel\_name ,

cl\_int \*errcode\_ret );

Выделить память под объект можно разными путями, их рассмотрение будет приведено в дальнейшем повествовании, сейчас же укажем только базовую функцию clCreateBuffer():

* context - это допустимый контекст OpenCL, используемый для создания буферного объекта.

cl\_mem clCreateBuffer(

cl\_context context,

cl\_mem\_flags flags,

size\_t size,

void\* host\_ptr,

cl\_int\* errcode\_ret);

* flags - это битовое поле, которое используется для указания информации о распределении и использовании, такой как область памяти, которая должна использоваться для выделения буферного объекта, и то, как он будет использоваться. В таблице « Флаги памяти» описаны возможные значения флагов . Если значение, указанное для flags, равно 0, используется значение по умолчанию CL\_​MEM\_​READ\_​WRITE.
* size - это размер в байтах выделяемого объекта буферной памяти.
* host\_ptr - это указатель на данные буфера, которые могут быть уже выделены приложением. Размер буфера, на который указывает host\_ptr, должен быть ≥ size байтов.

За передачу объектов памяти в аргументы функции ядра используется функция clSetKernelArg():

* kernel - допустимый объект ядра.

cl\_int clSetKernelArg(

cl\_kernel kernel,

cl\_uint arg\_index,

size\_t arg\_size,

const void\* arg\_value);

* arg\_index - индекс аргумента. Аргументы к ядру обозначаются индексами, которые идут от 0 для крайнего левого аргумента до n - 1, где n - общее количество аргументов, объявленных ядром (см. Ниже).
* arg\_size определяет размер значения аргумента. Если аргумент - объект памяти, размер - это размер объекта памяти. Для аргументов, объявленных с localквалификатором, указанный размер будет размером в байтах буфера, который должен быть выделен для local аргумента. Если аргумент имеет тип sampler\_t , значение arg\_size должно быть равно sizeof(cl\_sampler). Если аргумент имеет тип queue\_t , значение arg\_size должно быть равно sizeof(cl\_command\_queue). Для всех остальных аргументов размер будет размером типа аргумента.
* arg\_value - это указатель на данные, которые следует использовать в качестве значения аргумента для аргумента, указанного в arg\_index . Данные аргумента, на которые указывает arg\_value , копируются, и поэтому указатель arg\_value может быть повторно использован приложением после возврата clSetKernelArg . Указанное значение аргумента - это значение, используемое всеми вызовами API, которые ставят ядро в очередь ( clEnqueueNDRangeKernel ), пока значение аргумента не будет изменено вызовом clSetKernelArg для ядра .

Пример вызова этих функций будет приведен при рассмотрении разработки самого приложения. Сейчас уточним, что второй этап на этом заканчивается — создан объект ядра, выделена необходима память и в аргументы функции ядра переданы определенные значения.

**На третьем этапе,** происходит создание очереди команд — это механизм, посредством которого экземпляр ядра ставится в очередь экземпляром ядра, запущенным на устройстве без прямого участия хост-программы. В фреймворке предусмотрен особый тип данных cl\_command\_queue, за его инициализация отвечает функция clCreateCommandQueueWithProperties():

* context - должен быть допустимым контекстом OpenCL.

cl\_command\_queue clCreateCommandQueueWithProperties(

cl\_context context,

cl\_device\_id device,

const cl\_queue\_properties\* properties,

cl\_int\* errcode\_ret);

* device - должно быть устройством или подустройством, связанным с контекстом . Он может быть либо в списке устройств и подустройств, указанных при создании контекста с помощью clCreateContext, либо быть корневым устройством с тем же типом устройства, который указан при создании контекста с помощью clCreateContextFromType .
* properties - задает список свойств для очереди команд и их соответствующие значения. За каждым именем свойства сразу следует соответствующее желаемое значение. Список завершается цифрой 0. Список поддерживаемых свойств описан в таблице ниже. Если поддерживаемое свойство и его значение не указано в свойствах , будет использоваться его значение по умолчанию. properties могут быть, NULLв этом случае будут использоваться значения по умолчанию для поддерживаемых свойств очереди команд.
* errcode\_ret – возвращает код ошибки.

На этом третий этап заканчивается.

На четвертом этапе, происходит выполнение ядра на устройстве OpenCL, для этого понадобятся объект очереди команд и контекста. Для запуска ядра используется функция clEnqueueNDRangeKernel():

* command\_queue - это допустимая очередь команд хоста. Ядро будет поставлено в очередь для выполнения на устройстве, связанном с command\_queue .

cl\_int clEnqueueNDRangeKernel(

cl\_command\_queue command\_queue,

cl\_kernel kernel,

cl\_uint work\_dim,

const size\_t\* global\_work\_offset,

const size\_t\* global\_work\_size,

const size\_t\* local\_work\_size,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event\* event\_wait\_list,

cl\_event\* event);

* kernel - допустимый объект ядра. Контекст OpenCL, связанный с ядром и очередью команд, должен совпадать.
* work\_dim - это количество измерений, используемых для определения глобальных рабочих элементов и рабочих элементов в рабочей группе. work\_dim должен быть больше нуля и меньше или равен CL\_​DEVICE\_​MAX\_​WORK\_​ITEM\_​DIMENSIONS. Если global\_work\_size равно NULLили значение в любом переданном измерении равно 0, тогда команда ядра будет тривиально успешна после того, как будут удовлетворены ее зависимости событий, и впоследствии обновит свое событие завершения. Поведение в этой ситуации похоже на поведение маркера в очереди, за исключением того, что, в отличие от маркера, ядро ​​в очереди без событий, переданных в event\_wait\_list, может работать в любое время.
* global\_work\_offset может использоваться для указания массива значений без знака work\_dim, которые описывают смещение, используемое для вычисления глобального идентификатора рабочего элемента. Если global\_work\_offset равно NULL, глобальные идентификаторы начинаются со смещения (0, 0, 0).
* global\_work\_size указывает на массив значений без знака work\_dim, которые описывают количество глобальных рабочих элементов в измерениях work\_dim, которые будут выполнять функцию ядра. Общее количество глобальных рабочих элементов вычисляется как global\_work\_size [0] ×… × global\_work\_size [ work\_dim — 1].
* local\_work\_size указывает на массив значений без знака work\_dim, которые описывают количество рабочих элементов, составляющих рабочую группу (также называемую размером рабочей группы), которая будет выполнять ядро, указанное ядром . Общее количество рабочих элементов в рабочей группе вычисляется как local\_work\_size [0] ×… × local\_work\_size [ рабочий\_дим - 1]. Общее количество рабочих элементов в рабочей группе должно быть меньше или равно CL\_​KERNEL\_​WORK\_​GROUP\_​SIZEзначению, указанному в таблице запросов устройства объекта ядра , и количеству рабочих элементов, указанных в local\_work\_size [0],…, local\_work\_size [ work\_dim- 1] должно быть меньше или равно соответствующим значениям, указанным в CL\_​DEVICE\_​MAX\_​WORK\_​ITEM\_​SIZES[0],…, CL\_​DEVICE\_​MAX\_​WORK\_​ITEM\_​SIZES[ work\_dim - 1]. Явно указанный local\_work\_size будет использоваться для определения того, как разбить глобальные рабочие элементы, указанные global\_work\_size, на соответствующие экземпляры рабочей группы.
* event\_wait\_list и num\_events\_in\_wait\_list определяют события, которые необходимо завершить перед выполнением этой конкретной команды. Если event\_wait\_list равно NULL, то эта конкретная команда не ожидает завершения какого-либо события. Если event\_wait\_list равен NULL, num\_events\_in\_wait\_list должен быть 0. Если event\_wait\_list нет NULL, список событий, на которые указывает event\_wait\_list, должен быть действительным, а num\_events\_in\_wait\_list должен быть больше 0. События, указанные в event\_wait\_list, действуют как точки синхронизации. Контекст, связанный с событиями в event\_wait\_list иcommand\_queue должен быть таким же. Память, связанная с event\_wait\_list, может быть повторно использована или освобождена после возврата из функции.
* event возвращает объект события, который идентифицирует этот конкретный экземпляр ядра. Объекты событий уникальны и могут использоваться для идентификации конкретного экземпляра ядра позже. Если событие есть NULL, то для этого экземпляра ядра не будет создано никаких событий, и поэтому приложение не сможет запросить или поставить в очередь ожидания для этого конкретного экземпляра ядра. Если event\_wait\_list и аргументы события нет NULL, аргумент события не должен ссылаться на элемент массива event\_wait\_list .

Данная функция отвечает не только за запуск ядра на целевом устройстве, но и за определение размеров глобальной и локальной группы, о которых речь пойдет в следующем разделе. Пока же данная функция производит запуск ядра для исполнения, на чем и заканчивается заключительный четвертый этап.

В качестве вывода отметим, что описание этапы должны быть выполнены программистом для запуск программы на устройстве OpenCL. Порядок следования этапов определяет путь, по которому необходимо выполнить те или иные операции на устройстве, их порядок и наличие жестко фиксированы.

## 3.**4**.3. Модель памяти

## 3.**4**.4. Модель программирования

## 3.**5**. Совместное использование OpenGL и OpenCL

Программы, использующие OpenGL для создания и обработки графической информации, используют как центральный процессор, так и видеокарту. На центральном процессоре происходит формирование вершин и цветов текстуры или решение уравнений. Графический процессор принимает информацию от центрального процессора, преобразует вершины, необходимые для изображения движения объекта.

Процессор конечно обрабатывает данные намного быстрее, чем его графический аналог, но копирование больших массив памяти из ОЗУ в память видеокарты снижает производительность программы. Для компенсации подобных затрат использует взаимодействие OpenCL и OpenGL.

Интегрируя OpenCL в приложения OpenGL возможно возложить большую часть вычислений на графический процессор или на другие устройства, совместимые с OpenCL. Например, можно кодировать ядра, которые инициализируют отсчеты сигнала и производят преобразование Фурье над отсчетами. Такой подход имеет большое преимущество, графический процессор не только хорошо подходит для многих типов математических операций, но и вычисления, ориентированные на использование графического процессора, избавляют т необходимости передавать большие объемы данных между центральным и графическим процессорами.

Библиотека OpenGL также предоставляет доступ к использованию графического процессора для проведения математических расчетов, при помощи кодирования шейдеров — программ, которые исполняются на графическом процессоре. Шейдеры во многом схожи с ядрами OpenCL, но ядра обладают существенными преимуществами:

1. Ядра могут вызывать более широкий спектр функций;
2. Ядра могут обращаться к локальной и приватно памяти для высокоскоростной передачи данных;
3. Ядра имеют процедуры синхронизации, которые позволяют им обмениваться данными между рабочими элементами.

Последний третий пункт имеет огромное значение. Вершинный шейдер может обращаться к только одной вершине за раз, фрагментарный шейдер может обращаться только к одному фрагменту за раз. Но ядро может получить доступ ко всем данным на устройстве и синхронизировать их обработку с помощью барьеров. Такое преимущество делает ядра гибкими в отношении типов операций, которые могут выполняться на графическом процессоре.

Приложения OpenGL упаковывают данные, используя три структуры данных: объекты буфера вершин (VBO), объекты текстуры и объекты буфера рендеринга. Точно также приложения OpenCL получают доступ к данным, используя две структуры данных: объекты буфера и объекты изображения. Фундаментальная концепция, лежащая в основе взаимодействия OpenGL-OpenCL заключается в том, что объекты памяти OpenCL могут совместно использовать данные со структурами данных OpenGL.

Чтобы настроить взаимодействие OpenGL-OpenCL в коде, необходимо последовательно выполнить три шага:

1. Создать контекст OpenCL (cl\_context) которые ссылается на текущий контекст OpenGL или общую группу.
2. Создать объекты памяти OpenCL (объекты буфера или объекты изображения) из объектов данных OpenGL (VBO, объекты текстуры и объекты буфера рендеринга).
3. Получить монопольный доступ к совместно используемым данным для ядра. После выполнения ядра необходимо освободить данные, что их можно было передать в ренедеринг.

## 3.**5**.1. Создание контекста OpenCL

Чтобы установить взаимодействие между OpenCL и OpenGL, контекст OpenCL должен быть создан со ссылкой на контекст OpenGL. Этот контекст служит мостом между операционной системой и графическим окном. Как только контекст станет активен, операционная система направит все операции рендеринга OpenGL в соответствующее окно.

Для создания такого контекста необходимо вызвать функцию из OpenCL API для создания контекста clCreateContext(), в качестве первого параметра необходимо передать указатель на массив данных типа cl\_context\_proterties. В этом массиве необходимо перечислить имена свойств, которые зависят от операционной системы. Для ОС Windows массив должен содержать следующий параметры:

* CL\_GL\_CONTEXT\_KHR – дескриптор контекста рендеринга OpenGL (HGLRC) для окна Windows.
* CL\_WGL\_HDC\_KHR – дескриптор контекста устройства (HDC) для окна.
* CL\_CONTEXT\_PLATFORM – cl\_platform структура связанная с контекстом.

Для доступа к первым двум параметрам необходимо использовать функцию wglGetCurrentContext(), которая возвращает дескриптор контекста рендеринга, типа HGLRC, и функцию wglGetCurrentDC(), которая возвращает дескриптор контекста устройства окна. Третий аргумент возможно получить вызовом функции clGetPlatfromIDs().

После установления этих параметров можно вызвать функцию clCreateContext для создания контекста и использовать другие функции из OpenCL API.

## 3.**5**.2. Обмен данными между OpenCL и OpenGL

Чтобы обеспечить взаимодействие OpenGL-OpenCL, объекты памяти должен быть созданы из данных OpenGL. Доступ к этим данным OpenGL можно получить в одной из трех форм:

* буферные объекты вершин (VBO) — содержать данные вершин, также координаты, цвета и векторы нормалей;
* текстурные объекты содержат данные текстуры в виде изображения;
* объекты renderbuffer содержат пиксели для отображения.

Создание объектов памяти было подробно рассмотрено в предыдущем разделе. Для создания объектов памяти, способные обмениваться данными с OpenGL, необходимо вызвать одну из 4-х функций: clCreateFromGLBuffer(), clCreateFromGLTexture2D(), clCreateFromGLTexture3D(), или clCrateFromGLRenderBuffer(). При этом объект памяти OpenCL должен быть создан после создания соответствующего объекта OpenGL.

Приложения OpenGL хранят свои объекты данных в памяти графического процессора. Комбинируя OpenGL и OpenCL, можно создать приложения, которое инициализирует данные на графическом ускорителе с использованием ядра OpenCL. Это важно, потому что данные больше не нужно передавать от центрального процессора к графическому процессору.

## 3.**5**.3. Синхронизация доступа к общим данным

Подпрограммы OpenGL и OpenCL могут обмениваться данными, но не могут получить к ним доступ одновременно. Например, если ядро OpenCL обрабатывает общие данные как буферный объект, вершинный шейдер OpenGL не может получить доступ к общим данным VBO.

Синхронизации между OpenGL и OpenCL стала возможной благодаря двум функциям OpenCL. Первая — это clEnqueuAcquireGLObjects, которая гарантирует, что ядро будет иметь монопольный доступ к данным. Вторая функция clEnqueuReleaseGLObjects, которая позволяет другим процессам, таким как средство визуализации OpenGL, получать доступ к данным.

Работа этих функций аналогична мьютексу. Они служат для блокировки и освобождения данных, определенных в параметре mem\_obects.

При использовании этих функций следует помнить о следующем:

* прежде чем получить блокировку данных, необходимо вызвать функцию glFinish, чтобы убедиться, что все подпрограммы OpenGL завершили свою работу;
* после снятия блокировки данных необходимо вызвать функцию clFinish, чтобы убедиться в том, что все подпрограммы OpenCL завершили свою работу.

Таким образом, фундаментальной концепцией взаимодействия OpenGL-OpenCL является совместное использование данных. Объекты памяти OpenCL создаются из объектов памяти OpenGL. Для преобразования данных средствами OpenCL необходимо получить монопольный доступ к данным, провести необходимые вычисления и затем отпустить данные, чтобы они могли дальше идти по конвейеру обработки.

# 4. РАЗРАБОТКА ПРОГРАММЫ

Перейдем от рассмотрения технических и программных средств к их использованию для разработки программного обеспечения. Целью данной главы является: описание функционала и описание поэтапного процесса разработки программного обеспечения.

## 4.1. Описание основного функционала программы

Для проведения расшифровки цифровых голограмм разрабатываемое программное обеспечение должно включать себя следующий функционал:

1. Инициализация и настройка параметров работы графического ускорителя под управлением OpenCL.
2. Загрузка изображения(-ий) в память графического ускорителя.
3. Обработка целевого изображения(-ий) в памяти графического ускорителя.
4. Управление изображением(-ями): сохранение из памяти графического ускорителя на жесткий носитель; добавление и удаление из рабочего списка; отображение в главном окне программы.
5. Подключение программы к цифровой камере.
6. Проведение съемки в режиме реального времени.
7. Возможность сделать снимок с камеры из программы.

Разработку программного обеспечения будет проводить в (число) этапа. На первом этапе создадим функционал для инициализации и управления графическим ускорителем. Второй этап отводится под разработку графического интерфейса пользователя и функционала для управления целевыми изображениями. На третьем этапе будет производиться функционала и графического интерфейса для подключения и управления цифровой камерой.

## 4.2. Разработка функционала для инициализации и управления графическим ускорителем

При обзоре архитектуры и работы фреймворка OpenCL упоминалось, что разработка программы для проведения вычислений на гетерогенных устройствах начинается с инициализации хоста — платформ и находящихся под их управлением физических устройств. Проводить инициализацию будем в классах DHOCLInit и DHOCLHard.

Ниже на листинге (номер!!!) представлено описание класса DHOCLHard. Основное назначение этого класса заключается в хранении указателя на структуру cl\_platform\_id из фреймворка OpenCL, которая хранит платформу, из массива структур cl\_device\_id, в которых хранятся устройства, под управлением платформы, а также переменную m\_Size в которой хранится количество устройств.

Как видно на листинге (номер!!!) класс DHOCLHard включает в себя только один конструктор с параметров. Реализация этого конструктора представлен на листинге (номер!!!). При вызове конструктора происходит присвоение полю m\_Platform значения переменной platform, преедаваемой в качестве параметра конструктора. Затем происходит инициализация устройств при помощи метода clGetDeviceIDs. В результате работы конструктора будет создан объект, хранящий в себе указатель на платформу и массива устройств OpenCL.

class DHOCLHard

{

public:

DHOCLHard(cl\_platform\_id);

~DHOCLHard();

cl\_uint GetCountDevices() const;

cl\_device\_id GetDevice(unsigned int index) const;

cl\_platform\_id GetPlatform() const;

cl\_device\_id operator[](int index) const;

private:

cl\_platform\_id m\_Platform;

cl\_device\_id\* m\_Devices;

cl\_uint m\_Size;

};

В случае ошибки в полю m\_Devices будет присвоено значение nullptr, для получения более подробной информации об ошибке следует обратиться (!!!).

DHOCLHard::DHOCLHard(cl\_platform\_id platform)

{

m\_Platform = platform;

m\_Size = 0;

cl\_int err = clGetDeviceIDs(m\_Platform, CL\_DEVICE\_TYPE\_ALL, 0, nullptr, &m\_Size);

if (err != CL\_SUCCESS || m\_Size == 0)

m\_Devices = nullptr;

m\_Devices = new cl\_device\_id[m\_Size];

err = clGetDeviceIDs(m\_Platform, CL\_DEVICE\_TYPE\_ALL, m\_Size, m\_Devices, nullptr);

if (err != CL\_SUCCESS)

m\_Devices = nullptr;

}

Очевидно, что класс DHOCLHard производит инициализация только устройств для конкретной платформы, в то время как инициализация хоста предполагает инициализацию платформ и устройств. Для этих целей был создан класс DHOCLInit, описание которого представлено на листинге (номер!!!). Основное назначения этого класс заключается в инициализации платформ и применение их для создания массива из классов DHOCLHard, в которых хранятся платформы и устройства.

Инициализация платформ происходит в методе Init класса DHOCLInit, представленных на листинге ниже.

class DHOCLInit

{

public:

DHOCLInit() = default;

~DHOCLInit() = default;

bool Init();

std::size\_t GetSize() const;

const DHOCLHard& GetHardware(unsigned int index) const;

const DHOCLHard& operator[](int index) const;

private:

std::vector<DHOCLHard> m\_Hardware;

};

Создадим локальные переменные platfroms, указатель на тип cl\_platform\_id и num\_platforms, типа cl\_uint. В первой пременной будут храниться платформы, доступные для фреймворка OpenCL на текущем компьютере, во второй переменной будет храниться количество платформ. Инициализация этих переменных происходит функцией clGetPlatfromIDs. После инициализации локальных переменных проведем инициализацию поля m\_Hardware класса DHOCLInit.

bool DHOCLInit::Init()

{

cl\_platform\_id\* platfroms = nullptr;

cl\_uint num\_platfroms = 0;

cl\_int err = clGetPlatformIDs(0, platfroms, &num\_platfroms);

if (err != CL\_SUCCESS || num\_platfroms == 0)

return false;

platfroms = new cl\_platform\_id[num\_platfroms];

err = clGetPlatformIDs(num\_platfroms, platfroms, NULL);

if (err != CL\_SUCCESS)

return false;

for (int i = 0; i < num\_platfroms; i++)

{

DHOCLHard hard(platfroms[i]);

m\_Hardware.push\_back(hard);

}

return true;

}

## 4.3. Разработка пользовательского интерфейса

## 4.4. **Разработка программ для графического процессора**

# 5. ТЕСТИРОВАНИЕ

# ЗАКЛЮЧЕНИЕ

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК