# Оглавление

[​ ВВЕДЕНИЕ 3](#__RefHeading___Toc3336_2968763079)

[​ 1. ОБЗОР МАТЕМАТИЧЕСКИХ АЛГОРИТМОВ ДЛЯ РАСШИФРОВКИ ЦИФРОВЫХ ГОЛОГРАММ 5](#__RefHeading___Toc3338_2968763079)

[​ 2. ИСПОЛЬЗОВАНИЕ ГРАФИЧЕСКОГО ПРОЦЕССОРА ДЛЯ ПРОВЕДЕНИЯ МАТЕМАТИЧЕСКИХ РАСЧЕТОВ 6](#__RefHeading___Toc3340_2968763079)

[​ 2.1. Устройство и принцип работы графического ускорителя 6](#__RefHeading___Toc3342_2968763079)

[2.1.1 Конвейер инструкций 7](#__RefHeading___Toc3344_2968763079)

[​ 2.1.2. Управление графическим ускорителем 8](#__RefHeading___Toc3346_2968763079)

[​ 2.2. Возникновение и развитие технологии GPGPU 10](#__RefHeading___Toc3348_2968763079)

[3. ОБЗОР ФИЗИЧЕСКИХ УСТРОЙСТВ И ИСПОЛЬЗУЕМЫХ ПРОГРАММНЫХ ТЕХНОЛОГИЙ 13](#__RefHeading___Toc3350_2968763079)

[​ 3.1. Обзор используемых аппаратных и программных средств 13](#__RefHeading___Toc3352_2968763079)

[​ 3.2. Описание библиотеки MFC 13](#__RefHeading___Toc3354_2968763079)

[​ 3.2.1. Общая структура программы на базе библиотеке MFC 15](#__RefHeading___Toc3356_2968763079)

[​ 3.4.2.1. Класс CObject 16](#__RefHeading___Toc3358_2968763079)

[​ 3.4.2.2 Класс CCmdTarget 17](#__RefHeading___Toc3360_2968763079)

[​ 3.4.2.3 Классы CWnd 18](#__RefHeading___Toc3362_2968763079)

[​ 3.4.2.4 Архитектура «документ-представление» 24](#__RefHeading___Toc3364_2968763079)

[​ 3.3. Обзор спецификации OpenGL 25](#__RefHeading___Toc3366_2968763079)

[​ 3.4. Обзор фреймворка OpenCL 26](#__RefHeading___Toc3368_2968763079)

[​ 3.4.1 Модель платформы 27](#__RefHeading___Toc3370_2968763079)

[​ 3.4.2 Модель исполнения 35](#__RefHeading___Toc3372_2968763079)

[​ 3.4.3. Модель памяти 46](#__RefHeading___Toc3374_2968763079)

[​ 3.4.4. Модель программирования 51](#__RefHeading___Toc3376_2968763079)

[​ 3.5. Совместное использование OpenGL и OpenCL 54](#__RefHeading___Toc3378_2968763079)

[​ 3.5.1. Создание контекста OpenCL 55](#__RefHeading___Toc3380_2968763079)

[​ 3.5.2. Обмен данными между OpenCL и OpenGL 56](#__RefHeading___Toc3382_2968763079)

[​ 3.5.3. Синхронизация доступа к общим данным 57](#__RefHeading___Toc3384_2968763079)

[​ 4. РАЗРАБОТКА ПРОГРАММЫ 59](#__RefHeading___Toc3386_2968763079)

[​ 4.1. Описание основного функционала программы 59](#__RefHeading___Toc3388_2968763079)

[​ 4.2. Использование фреймворка OpenCL 59](#__RefHeading___Toc3390_2968763079)

[​ 4.2.1. Инициализация платформ и устройств OpenCL 60](#__RefHeading___Toc3392_2968763079)

[​ 4.2.2. Инициализация структур данных OpenCL для исполнения программ 65](#__RefHeading___Toc3394_2968763079)

[​ 4.3. Разработка графического интерфейса пользователя 65](#__RefHeading___Toc3396_2968763079)

[​ 5. ТЕСТИРОВАНИЕ 66](#__RefHeading___Toc3398_2968763079)

[​ ЗАКЛЮЧЕНИЕ 67](#__RefHeading___Toc3400_2968763079)

[​ БИБЛИОГРАФИЧЕСКИЙ СПИСОК 68](#__RefHeading___Toc3402_2968763079)

[​ ПРИЛОЖЕНИЕ 69](#__RefHeading___Toc3404_2968763079)

# ВВЕДЕНИЕ

Голография – это метод регистрации произвольного колебательного процесса, позволяющий зафиксировать как амплитуду, так и фазу колебания, а затем воспроизвести их в любой удобный момент времени.

По результатам обработки голограммы можно судить об изменении исследуемого объекта во времени. При помощи этого можно создать модель тех изменений, которые повлияли состояние объекта. Для проведения анализа применяют метод пошагового вазового сдвига и преобразование Френеля.

Направление является перспективным, но для проведения экспериментов и исследований в этой области необходимо обрабатывать большие массивы данных за короткий промежуток времени. Достичь этого можно только при использовании аппаратных и программных средств вычислительной техники. В случае с аппаратных обеспечением на сегодняшний день создано большое количество процессоров с большими тактовыми частотами. С программными средствами дела обстоят не лучшим образом. На сегодняшний ведется активная разработка необходимого программного обеспечения, но ни одно из них не может удовлетворить потребности конечного пользователя в полной мере.

В данной работе разрабатывается программное обеспечение реализующее алгоритм расшифровки цифровых голограмм. Для ускоренной обработки информации будет использоваться технология GPGPU. Суть данной технологии заключается в использовании графического процессора для проведения математических вычислений. Графический процессор позволяет использовать большое число вычислительных элементом — потоков для обработки больших массивов данных. Данный подход позволит значительно увеличить скорость обработки данных.

Тестирование разработанного программного обеспечения будет проводить на экспериментальной установке. Считывание информации происходит при помощи фотоаппарата фирмы Cannon 650D. Обработкой полученной информации занимается видеокарта.

**Актуальность работы** заключается в модификации аналогичного программного обеспечения для повышения скорости обработки данных. Данное программное обеспечение позволяет производить расшифровку цифровых голограмм в реальном времени.

**Новизна научной работы** заключается в исследовании основных алгоритмов расшифровки цифровых голограмм. Разработан алгоритм преобразования Френеля и PSI преобразования с использованием технологии GPGPU для проведения математических преобразований

**Цель работы**: разработка компьютерной системы для восстановления изображения из зарегистрированных голограмм в реальном времени на языке программирования C++.

Для достижения поставленной цели необходимо решить следующие **основные задачи**:

1. Анализ существующих решений
2. Выбор алгоритма для расшифровки цифровых голограмм
3. Реализация алгоритмов для графического процессора
4. Тестирование

# 1. ОБЗОР МАТЕМАТИЧЕСКИХ АЛГОРИТМОВ ДЛЯ РАСШИФРОВКИ ЦИФРОВЫХ ГОЛОГРАММ

Голография – это процесс регистрации интерференционной картины - голограммы, которая образована волной, отраженной некоторым объектом, освещаемым источником света (объектная волна), и когерентной с ней волной, идущей непосредственно от источника света (опорная волна).

При помощи голографии можно проследить изменение того или иного объекта исследования под воздействием излучения. В дальнейшем полученные данные используется для построения математической модели процесса изменения исследуемого объекта. Данное направление позволяет исследовать промышленные детали и конструкции в заводских условиях.

Процесс регистрации голограммы во многом похож на процесс фотографии. В том и в другом случае используется светочувствительная среда, которая регистрирует интенсивность светового поля. Но между этими процессами есть одно важное различие. На фотографии записывается только интенсивность объектной волны, а на голограмме регистрируется информация о полной волне – амплитуды и фазы.

Для регистрации голограммы требуется высокое разрешение светочувствительной среды, использование которой возможно только в лабораторных условиях. В связи с этим для регистрации голограмм используют обычную матрицу фотоаппарата. Ее разрешение значительно ниже, но позволяет зарегистрировать цифровую голограмму.

Основной особенностью цифровой голографии является то, что анализ, синтез и математическое преобразование происходит за счет средств вычислительной техники. Но для качественного проведения анализа цифровых голограмм необходимо обрабатывать большое количество цифровых голограмм за короткий промежуток времени. Центральный процессор компьютера нацелен больше на быстрое исполнение инструкций и не предоставляет возможностей для большого распараллеливания программ. Таким образом, в разрабатываемом приложении используется графический процессор, архитектура которого нацелена на параллелизм исполняемых инструкций. Такой подход позволяет значительно понизить временные траты на расшифровку цифровых голограмм.

## 1.1. Моделирование голографического процесса

Рассмотрим схему на рисунке 1, иллюстрирующую процесс регистрации и расшифровки цифровых голограмм.

|  |
| --- |
|  |
| Рис. 1. Координатная система получения и расшифровки цифровой голограммы |

Плоскость (x,y) – плоскость касательная к объекту или достаточно близкая к нему и параллельна плоскости исследуемого объекта. Световая волна, отраженная от исследуемого объекта аппроксимируется по законам геометрической оптики. Плоскость (ξ,η) – плоскость формирования голограммы. Плоскость (xʹ,yʹ) – плоскость регистрации голограммы.

Рассмотрим подробнее шаги получения и расшифровки цифровых голограмм.

Шаг 1. Задаем массив комплексных чисел размером N

Шаг 2. Пересчет волнового фронта и плоскости объекта в плоскость голограммы в зависимости от расстояния d. Данный шаг выполняется при помощи преобразования Фурье или Френеля. Обсуждение преобразования Френеля приведено ниже в разделе 1.3.

Шаг 3. Добавляем опорный фронт к объектному пучку для получения голограммы.

Шаг 4. Формируем волновое поле сразу же за голограммой умножением поля голограммы на волну, соответствующую опорной.

Шаг 5. Для пересчета в плоскость изображения снова делаем преобразование Френеля над волновым полем сразу же за голограммой. В плоскости изображения определяется комплексная амплитуда волнового фронта.

Таким образом, алгоритм расшифровки цифровых голограмм состоит из двух этапов. Первый этап реализуется методом пошагового фазового сдвига. Второй этап заключается в использовании преобразования Френеля. Рассмотрим оба этапа подробнее.

## 1.2. Метод пошагового фазового сдвига

При использовании метода цифровой голографии строится математическая (комплексная) модель голограммы, содержащая в себе амплитуду и фазу исследуемого объекта.

Значение амплитуды A и разности фаз Δφ рассчитывается по нескольким цифровым голограммам. Для определения фазы используется следующая формула:

где I = (I0,…,Im-1)T – вектор интенсивностей, рассчитанный при различных углах фазовых сдвигов; C = (cosδ0,…,cosδm-1)T, S = (sinδ0,…,sinδm-1)T – векторы косинусов и синусов для данных углов; I┴ - вектор, ортогональный вектору I.

Ортогональный вектор рассчитывается по следующему матричному уравнению:

где квадратная матрица M при размере m задается следующим образом:

Амплитуда определяется по следующей формуле:

Определив значение фазы и амплитуды можно записать выражение для математической голограммы. Данное волновое поле получается умножением волнового поля голограммы на волну, соответствующую опорной.

## 1.3. Преобразование Френеля

# 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

В процессе развития технологии обработки графической информации, конструкция графического процессора претерпела значительные изменения. На сегодняшний день, графический ускоритель или видеокарта, представляет собой отдельное вычислительное устройство. Размещаются графические процессоры на отдельной плате, которая соединяется с материнской платы через порт PCI. Видеокарта имеет свою систему охлаждения и специализированную оперативную память — видеопамять, в которой хранятся обрабатываемые данные.

Основное назначение графического ускорителя - параллельная обработка больших массивов данных. В связи с чем, архитектурное устройство всех графических процессоров имеет общие черты. Обязательным является система из параллельных вычислительных устройств, которые называются ядрами (kernel). Такое архитектурное решение позволяет обрабатывать большие массивы данных одновременно и независимо друг от друга. То есть позволяет распараллелить вычисления по большим массивам данных.

Для кодирования таких алгоритмов была придумана специальная концепция - Single Instruction - Multiple Data (одна инструкция для множества данных) или сокращенно SIMD. Основной характеристикой такой концепции является потоковая обработка данных. Вычислительное устройство или процессор, который работает по принципу SIMD, преобразует поток данных в поток результатов, используя программу как функцию преобразования.

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

## 2.1.1 Конвейер инструкций

Помимо параллельной обработки данных присутствует и другая архитектурная особенность видеокарт, обусловленная специфичными потребностями компьютерной графики. Структура проведения операция на всех современных графических процессорах может быть представлена конвейером. Устройство такого конвейера разработано 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, операции исполняются одновременно и не зависят друг от друга.

Рассмотрим подробнее работу конвейера инструкций. В приложении (указать номер!!!) представлена графическая иллюстрация принципа конвейерной обработки данных.

**Первый этап** - определение состояния объектов в сцене. Почти все операции на данном этапе выполняет центральный процессор. По окончанию работы результат пересылается драйвером в графический чипсет.

**Второй этап -** декомпозиция (разделение на примитивы) геометрических моделей. На экране отображаемый объект выводится через геометрический примитивы. В роли таких примитивов выступает треугольник и обычная плоская фигура — полигон.

**Третий этап** — трансформация и расчет освещености. К вершинам геометрических примитивов применяют различные эффекты преобразований и освещенности. Данный этап выполняется во многом аппаратными средствами, но при помощи шейдеров можно проводить изменения динамически. В конце параметры вершин нормализуются и приводятся к целочисленному виду.

**Четвертый этап** - установка примитивов. На протяжении предыдущих этап графический процессор обрабатывал вершины по отдельности. На данном этапе происходит «собрка» существующих вершин в треугольники, определение цвета и отсечение невидимых областей.

**Пятый этап** - текстурирование. На данном этапе происходит наложени текстур. Исполнение шейдеров невозможно. Такой процесс называется рендерингом.

**Шестой этап** - заключительный этап. На данном этапе происходит удаление дефектов. В добавок к этому происходит применение какого либо эффекта к сформированному изображению.

В зависимости от типа видео-ускорителя часть этапов просчитывается программными средствами, а другая часть – аппаратными средствами. Более современные графические процессоры исполняют часть или все части операций графического конвейера аппаратными средствами.

## 2.1.2. Управление графическим ускорителем

Управление графическим процессором происходит через центральный процессор. При помощи встроенных программ ЦП выполняет за программиста всю рутинную работу, такую как выделение памяти и представление данных в необходимом формате.

Также существует и драйвер графического процессора — программа, которая непосредственно управляет как самим GPU, так и его памятью. Драйвер работает как часть операционной системы - он получает от других приложений запросы — задачи для графического процессора, после чего передает их на исполнение GPU. Функции драйвера:

* распределение ресурсов графического процессора между несколькими приложениями;
* загрузка данных в видеопамять и регистры графического процессора, копирование данных из видеопамяти в оперативную память компьютера;
* загрузка в графический процессор и запуск на исполнение пользовательских программ;
* автоматическое распределение расчетов между параллельными вычислительными блоками графического процессора; исполнение стандартных алгоритмов обработки графики;
* регулирование частоты графического процессора для поддержания рабочей температуры.

Как правило драйвер предоставляется разработчиком видеокарты, так как это требует детального знания их технического устройства и особенностей. Они обновляются для поддержки процессоров новых моделей.

Пользовательское приложение может само обращаться к драйверу графического адаптера напрямую. Но для этого разработчику такой программы придется включить в код программы большое количество стандартных операций управления графическим процессором. Для автоматизации и упрощения работы с драйвером видеокарты были разработаны специальные программные инструменты, которые берут на себя все рутинный операции. Такие инструменты называются программный интерфейс приложений (application programming interface, или API). Такой интерфейс может поставляться самим разработчиков драйвера, либо сторонними разработчиками. В качестве примера такого интерфейса выступает OpenCL. Подробнее эти интерфейс буду рассмотрены в главе 3.

Любое приложение, которое использует графический процессор, должно состоять из двух частей. Первая часть исполняется на центральном процессоре. Для ее создания можно использовать любой известный язык программирования. Вторая часть исполняется на самом графическим процессором. Для ее создания необходимо использовать специальныя языки программирования. В случае с OpenCL это язык программирования OpenCL C. Программы создаются в отдельных файла, компилируются и запускаются на исполнением графическим процессором средствами OpenCL.

## 2.2. Возникновение и развитие технологии GPGPU

Важным критерием любого вычислительного устровта является производительность - количественную характеристику скорости выполнения определенных операций на вычислительном устройстве. Зачастую производительность определяется над вещественными числами (floating-point operations per seconds, FLOPS).

Выделяют пиковую и реальную производительность. Первая используется при учете только скорости расчета, а вторая при влиянии других факторов, таких как память.

Производительность компьютера зависит от центрального процессора и памяти. В идеальном случае, процессоры с высокой тактовой частотой и большим объемом интегрированной памяти могли бы обладать высокой производительностью. Но из-за сложности производства подобных чипов их цена увеличивается в разы. Из-за этого основной объект находится в отдельных модулях — оперативной памяти. Таким образом, реальная производительность вычислительной системы во много раз ниже пиковой.

По мере развития компьютерной индустрии, разработчики стараются объединить большой объем памяти и центральной процессор в один чип. Так, например, современная видеокарта представляет собой отдельную плату с собственными вычислительными элементами и интегрированной памятью.

История развития графического процессора насчитывает большое количество различных этапов. Основным толчком в развитии данной технологии, послужило создание графического конвейера и шейдеров. Все вместе существенное увеличило скорость обработки графической информации. Промежуточным итогом можно считать тот факт, 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) для выполнения вычислений в приложениях, которые традиционно обрабатываются центральный процессор. Подоюная технология способна значительно увеличить производтельность алгоритма обработки данных.

С другой стороны, технология GPGPU предоставляет возможность обработки больших массивов данных между центральным и графическим процессором. При этом данные можно представлять как в виде обычных целочисленных или вещественных массивов чисел, либо в виде текстуры — изображения. Конечно, графический процессор обладает меньшей тактовой частотой чем традиционный центральный процессор, но этот недостаток компенсируется большим количеством ядер-обработчиков. За счет этого, графические процессоры могут обрабатывать гораздо больше изображений и графических данных в секунду, чем традиционные процессоры. Поэтому, перенос данных в графическую форму (в виде изображения), может существенно ускорить обработку данных.

В принципе, любая произвольная логическая функция, включая функции сложения, умножение и другие математические функции могут быть построены из функционально полного набора логических элементов. Вычисления общего назначения на графических процессорах стали более практичными и популярными с появлением как программируемых шейдеров.

По мере развития самой технологии, развивался и интерфейс взаимодействия с графическим процессором. Была создана Nvidia CUDA и OpenCL от Apple / Khronos Group. Это позволяет использовать современные конвейеры GPGPU могут использовать скорость графического процессора, не требуя полного и явного преобразования данных в графическую форму.

Подведем итог всего выше сказанного. Графический процессор эволюционировал от простого устройств обработки графики, до полноценного вычислительного устройства, со своей системой охлаждения и интерфейсом. Основным отличием его от обычного центрального процессора является большое количество ядер или обрабатывающих элементов, при этом тактовые частоты значительно ниже, чем у центрального процессора. Такое устройство графического процессора позволяет быстрее обрабатывать большие массивы данных, за счет распараллеливания алгоритма обработки. Таким образом, при переносе вычислительной задачи с центрального процессора на графический стоит иметь в виду, что придется приспособить алгоритм обработки под архитектуру графического процессора.

# 3. ОБЗОР ФИЗИЧЕСКИХ УСТРОЙСТВ И ИСПОЛЬЗУЕМЫХ ПРОГРАММНЫХ ТЕХНОЛОГИЙ

## 3.1. Обзор используемых аппаратных и программных средств

Начнем с обзора аппаратных средств целевого компьютера и программных технологий, используемых при реализации приложения.

## 3.2. Описание библиотеки MFC

Разработка программного обеспечения под операционную систему (ОС) Windows, любой версии, не обходится без использования фреймворка Win32 API. Эта технология позволяет программистам создавать графический интерфейс для своих программ и использовать прочие ресурсы ОС Windows, такие как процессы и потоки, систему ввода-вывода и так далее. Но разработка программ с использованием Win32 API 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 специальные библиотеки, которые упрощают процесс создания графического интерфейса и прочего функционала. Одной из таких является библиотека MFC (Microsoft Foundation Classes), разработанная в объектно-ориентированной парадигме.

Библиотека MFC состоит из классов, каждый из которых предоставляет определенный функционал для работы с ОС Windows. При этом библиотека берет на себя выполнение большей части черной работы. Что позволяет разработчику программного обеспечения сосредоточиться на реализации своего продукта. Помимо этого библиотека предоставляет новые механизмы, которые упрощаю разработку программного обеспечения для ОС Windows.

Библиотека классов MFC охватывает все компоненты Windows – окна, блоки диалога, контексты устройств, общие объекты GDI (битовые образы и кисти), элементы управления и многие другие стандартные элементы. Для управления приложения в операционной системе Windows используются сообщения. Сообщение – это структура данных, которая генерируется операционной системой и передается приложению каждый раз, когда происходит какое-то зарезервированное приложением событие; например, нажатие пользователем на мышку или клавиатуру, перемещение главного окна приложения, взаимодействие с элементов управления и прочее. При использовании Win32 API программист должен самостоятельно позаботиться об обработке сообщений от того или иного элемента управления. Вследствие 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 источнику (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 программ, написанных с использованием библиотеке, MFC является класс CwinApp. Его структура (поля и методы) зависит от конкретного приложения, но есть и общие поля и методы. Данный класс выполняет одну из основных частей работы программы в системе Windows, которые программист должен был реализовывать самостоятельно при программировании на Win32 API. К одной из таких задач относится создание процесса приложения в системе; создание окна приложения; обработка сообщение от ОС Windows. Таким образом может быть создан только один объект класса WinApp в проектируемом приложении. Зачастую программист создает собственный класс для запуска своего приложения в системе, для этого ему необходимо наследоваться от класса CWinApp и дополнить его поля и методы на свое усмотрение. Пример подобного класса будет рассмотрен в дальнейшем при обсуждении процесса разработки приложения.

При использовании Win32 API программист должен был самостоятельно определять функцию WinMain и многие другие строки вспомогательного кода. Но при использовании библиотеки MFC эту работу на себя берет класс WinApp. Для инициализации приложения используется метод класса InitApplication и InitInstance. Первый метод InitApplication выполняет инициализацию на уровне приложения. Второй метод InitInstance выполняет инициализацию каждой запущенной копии приложения, то есть создает главное окно приложения.

Помимо инициализации приложения и создания главного окна класс берет на себя обработку сообщений от операционной системы. Эту задачу выполняет метод Run. Этот метод может быть также переопределен разработчиком целевого программного обеспечения.

При завершении работы приложения вызывается метод ExitInstance, который заканчивает работу и цикл обработки сообщений.

Ранее было отмечено, что библиотека MFC выполнена в объектно-ориентированной парадигме программирования. Большинство функциональных возможностей Win32 API были спроектированы в классы. Также разработчики библиотеки добавили новые функциональные возможности для разработки программного обеспечения. Таким образом, библиотека получилась очень большой. Обзор всех возможностей библиотеки занимает большое количество времени и места на бумаге. Поэтому в данном разделе обсудим лишь основополагающие концепции, заложенные разработчиками библиотеки, иерархию классов в общем представлении и несколько программных механизмов, используемые в процессе разработки проекта.

## 3.4.2.1. Класс CObject

В приложении (указать номер!!!) представлена графическая иллюстрация иерархии классов библиотеки MFC. На вершине иерархии располагается класс CObject. Класс CObject предоставляет другим классам функционал по сохранения и восстановлению объектов. Другими словами состояние объекта может быть записано в файл на диске, а затем из файла возможно восстановить сохраненный объект — сериализация (serialization).

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.

В классе CObject хранится одно единственное поле classObejct типа CRuntimeClass. В данном поле хранится информация об объекте класса. Это поле используется для определения принадлежности объекта конкретному классу. Для этого необходимо использовать метод IsKindOf. Эта функция выполняет безопасное приведение типа объекта. Аргументом функции выступает 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!!!).

## 3.4.2.2 Класс CCmdTarget

Непосредственно от класса CObject наследуется ряд других классов, которые сами являются базовыми для остальных классов библиотеки MFC. В первую очередь это класс CCmdTarget или CCmdTargetEx, представляющий основу структуры любого приложения. Основной особенностью класса CCmdTarget и потомков этого класса является то, что объекты этих классов могут получать от операционной системы сообщения и обрабатывать их.

В приложении (указать номер!!!) представлен фрагмент иерархии классов библиотеки MFC. Данный фрагмент демонстрирует потомков класс CCmdTarget, которые используются для запуска и создания оконного приложения Windows. Кратко опишем назначение классов-наследников от CCmdTarget.

От класса CCmdTarget наследуется класс CWinThread (CWinThreadEx), представляющий подзадачи приложения. Эта подзадача, называемая главной, представляется классом CWinApp (CWinAppEx), наследованным от класса CWinThread.

Большинство приложений работают с данными или документами, хранимыми на диске в отдельных файлах. Класс CDocument (CDocumentEx), наследованный от базового класса CCmdTarget, служит для представления документов приложения.

Еще один важный класс, наследуемый от CCmdTarget, называется CDocTemplate (CDocTemplateEx). От этого класса наследуются два класса CSingleDocTemplate (CSinglDocTemplateEx) и CMultiDocTemplate (CMultiDocTemplateEx). Все эти классы предназначены для синхронизации и управления основными объектами представляющими приложение - окнами, документами и используемыми ими ресурсами.

## 3.4.2.3 Классы CWnd

Практически все приложения имеют пользовательский интерфейс, построенный на основе окон. Это может быть диалоговая панель, одно окно или несколько окон, связанных вместе. Основные свойства окон представлены классом CWnd (CWndEx), наследованным от класса CCmdTarget. Но создавать объект данного класса, как правило, не требуется. Вместо этого используются потомки класса CWnd (CWndEX), через которые реализованы различные оконные приложения Windows.

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.

Рассмотрим подробнее наследников класса CWnd.

Первым в очереди располагается класс CFrameWnd (CFrameWndEx). Данный класс представляет обрамляющие окна (frame window), в том числе и главные окна приложения. 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. . От этого класса также наследуются классы CMDIChildWnd XE "CMDIChildWnd" и CMDIFrameWnd XE "CMDIFrameWnd" , используемые для отображения окон многооконного интерфейса MDI. Класс CMDIFrameWnd представляет главное окно приложения MDI, а класс CMDIChildWnd - его дочерние окна MDI. Класс CMiniFrameWnd XE "CMiniFrameWnd" применяется для отображения окон уменьшенного размера. Такие окна обычно используются для отображения в них панели управления.

Следующим рассмотренным классом будет CView и его наследники. Эти классы представляют окно просмотра документов приложения. Именно окно просмотра используется для вывода на экран документа, с которым работает приложения. Через это окно пользователь может изменять документ.

При разработке приложения программист должен наследовать собственные классы просмотра документов либо от базового класса CView, либо от одного из нескольких порожденных классов, определенных в библиотеке MFC.

Классы, наследованные от CCtrlView XE "CCtrlView" , используют для отображения документа готовые органы управления. Например, класс CEditView XE "CEditView" использует орган управления edit (редактор). Более подробно эти классы будут описаны позже, когда мы будем рассказывать о средствах автоматизированного программирования MFC AppWizard и ClassWizard.

Класс CScrollView XE "CScrollView" представляет окно просмотра, которое имеет полосы свертки. В классе определены специальные методы, управляющие полосами просмотра

Класс CFormView XE "CFormView" позволяет создать окно просмотра документа, основанное на диалоговой панели. От этого класса наследуются еще два класса CRecordView XE "CRecordView" и CDaoRecordView XE "CDaoRecordView" . Эти классы используются для просмотра записей баз данных.

Рассмотрев класс представления, перейдем к классу CDocument. Этот класс предоставлет базовые функциональные возможности для классов документов, определенных пользователем. Он поддерживает все стандартные операции, такие как создание документа, его загрузка и сохранение. Библиотека MFC работает с документами, используя интерфейс, определенный в классе CDocument.

Для реализации документа в типичном приложении необходимо проделать следующую последовательность действий:

* для каждого типа документа образовать класс на базе CDocument;
* добавить в него переменные для хранения всех данных документа;
* реализовать функции для чтения и модификации этих данных;
* переопределить функцию CObject::Serialize в новом классе документа для организации чтения/записи данных документа с диска на диск.

Данные документа определяются как переменные специального класса документа, производного от CDocument.

Кроме перечисленных классов от базового класса CWnd наследуются классы, управляющие диалоговыми панелями. Если вы желаете создать диалоговую панель, вы можете наследовать класс от CDialog.

Вместе с диалоговыми панелями обычно используется класс CDataExchange (CDataExchangeEx). Класс CDataExchange (CDataExchangeEx) обеспечивает работу процедур обмена данными DDX (Dialog Data Exchange) и проверки данных DDV (Dialog Data Validation) используемых для диалоговых панелей. В отличие от класса CDialog, класс CDataExchange не наследуется от какого-либо другого класса.

Когда вы создаете блокнот, состоящий из нескольких страниц, то каждая такая страница является объектом класса, наследованного от CPropertyPage (CPropertyPageEx).

От класса CDialog (CDialogEx) наследуется ряд классов, представляющих собой стандартные диалоговые панели для выбора шрифта, цвета, вывода документа на печать, поиска в документе определенной последовательности символов, а также поиска и замены одной последовательности символов другой последовательностью.

Чтобы создать стандартный диалог, вы можете просто определить объект соответствующего класса. Дальнейшее управление такой панелью осуществляется методами класса.

Рассмотрим дочерние классы CWnd, которые называются органы управления. Назначение таких классов заключается в создании интерфейса с которым будет взаимодействовать конечный пользователь приложения. Существует ряд органов управления, встроенных в операционную систему, к ним относятся: кнопки, полосы прокрутки, редакторы текста, переключатели т.д.

Для работы с этими органами управления в бибилиотеки MFC предусмотрены специальные классы:

|  |  |
| --- | --- |
| Класс | Назначение |
| CAnimateCtrl (CAnimateCtrlEx) | Используется для отображения видеоинформации |
| CBitmapButton (CBitmapButtonEx) | Кнопка с рисунком |
| CButton (CButtonEx) | Кнопка |
| CComboBox (CComboBoxEx) | Список с окном редактирования |
| CEdit (CEditEx) | Поле редактирования |
| CHeaderCtrl (CHeaderCtrlEx) | Заголовок для таблицы |
| CHotKeyCtrl (CHotKeyCtrlEx) | Предназначен для ввода комбинации клавиш акселераторов |
| CListBox (CListBoxEx) | Список |
| CListCrtl (CListCrtlEx) | Может использоваться для отображения списка пиктограмм |
| CProgressCtrl (CProgressCtrlEx) | Линейный индикатор |
| CPropertySheet (CPropertySheetEx) | Блокнот. Может состоять из нескольких страниц |
| CRichEditCtrl (CRichEditCtrlEx) | Окно редактирования, в котором можно редактировать форматированный текст |
| CScrollBar (CScrollBarEx) | Полоса просмотра |
| CSliderCtrl (CSliderCtrlEx) | Движок |
| CSpinButtonCtrl (CSpinButtonCtrlEx) | Обычно используется для увеличения или уменьшения значения какого-нибудь параметра |
| CStatic (CStaticEx) | Статический орган управления |
| CTabCtrl (CTabCtrlEx) | Набор “закладок” |
| CToolBarCtrl (CToolBarCtrlEx) | Панель управления |
| CToolTipCtrl (CToolTipCtrlEx) | Маленькое окно содержащее строку текста |
| CTreeCtrl (CTreeCtrlEx) | Орган управления, который позволяет просматривать иерархические структуры данных |

## 3.4.2.4 Архитектура «документ-представление»

Работу приложения, написанного под ОС Windows, можно разделить на три категории. Первая предоставляет пользователю внешний вид приложения в виде картинок. Вторая часть отслеживает нажатие кнопки на клавиатуре, перемещение курсора и т.д. И третья часть осуществляет чтение, запись и изменение данных, с которыми работает приложение. В рамках библиотеки MFC для каждой из трех категории существует свой собственный термин. Для первой категории используют термин представление - это окна, которые отображают данные документа и управляют взаимодействием пользователя с ними. Для второй фрейм - это окна, которое отвечает за отслеживание манипуляций пользователя и клавиатурой, мышкой и так далее. Для третьей категории используют термин документ - это данные, с которыми работает приложение.

Для представлений и фрейма создаются окна Windows, в то время как для документа, как правило, не создается отдельного окна, а ограничиваются только отдельным объектом класса документа в фрейме. Кроме того, количество объектов класса документа не ограничено. Фрейм может передавать данные от одного документа к другому. Эта особенность распространяется и на представления.

Архитектура «документ-представление» призвана облегчить программисту процесс создания оконного интерфейса Windows. Для этого, помимо специальных классов для документа, фрейма и представления, существует шаблон для взаимодействия этих трех составляющих, называется этот класс CdocTemplate. Этот абстрактный базовый класс, в котором реализованы основные функциональные возможности для работы с шаблонами документов – организация и управление взаимодействием между классами трех типов:

* классом документа, образованного из CDocument;
* классом представления, который изображает данные документа. Можно создать этот класс на базе CView;
* классом фрейма, который содержит представление.

## 3.3. Обзор спецификации OpenGL

В данном разделе обсудим основное назначение спецификации, устройство и принцип работы OpenGL. Для более подробного ознакомления стоит обратиться к официальной документации (указать!!!) или к (указать!!!).

OpenGL представляет собой универсальный программный интерфейс к графическому адаптеру. Универсальность данного интерфейса заключается в том, что его использует большинство графических устройств. Этот интерфейс разрабатывался давно, поэтому стиль кода состоит из вызова функций на традиционно языке программирования C. Интерфейс состоит из более чем 200 функций.

Библиотека OpenGL позволяет работать с графикой для большинства графических устройств для любой операционной системы. Поэтому библиотека не включает в себя функционал для работы с окнами и системой ввода-вывода. Также в библиотеки отсутствуют функции для описания сложных трехмерных объектов. Все эти операции необходимо выполнить самому программисту, используя те или иные программные средства.

OpenGL – это машина состояния. Через функции библиотеки мы можем задать цвет, размер и положение графического объекта. Для этого необходимо задать значение для конкретной переменной состояния. Для изменения состояния той или иной переменной стоит использовать функции glEnable() - начало изменения и glDisable() - конец изменения. При этом в качестве аргмента в первой функции необходиом передать саму переменную.

При работе на том или ином графическом адаптере OpenGL библиотека выполняет следующие операции:

1. создание сцены из графический примитивов (треугольников);
2. позиционирование объектов сцены;
3. наложение цвета;
4. растровая развертка.

Растеризация – это процесс преобразования геометрических и пиксельных данных во фрагменты. Каждый фрагмент соответствует пикселю в буфере кадра. Шаблоны линий и полигонов, толщина линии, размер точек, модель заливки, вычисления связанные с наложением для поддержки сглаживания принимаются в расчет при развертке двух вершин в линию или вычислении внутренних пикселей полигона. Каждый фрагмент имеет ассоциированные с ним значения цвета и глубины.

В процессе выполнения всех этих этапов библиотекой OpenGL могут производиться и другие операции. Например, удаление частей объектов, скрытых другими объектами.

Библиотека OpenGL предоставляет функционал для наложения текстур изображения на геометрические объекты, чтобы заставить их выглядеть более реалистично.

Таким образом, библиотека OpenGL предоставляет гибкий интерфейс программисту для работы с графическим адаптером.

## 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:

cl\_int clGetPlatformIDs(

cl\_uint num\_entries ,

cl\_platform\_id \*platforms ,

cl\_uint \*num\_platforms );

* num\_entries - это количество записей, которые можно добавить на platfroms. Если платформы нет , num\_entries должно быть больше нуля.
* platforms - возвращает список найденных платформ OpenCL. Эти значения, возвращаемые в платформах, могут быть использованы для идентификации конкретной платформы OpenCL. Если platforms есть, этот аргумент игнорируется. Возвращаемое количество платформ OpenCL является минимальным из значения, указанного в num\_entries, или количества доступных платформ OpenCL.
* num\_platfroms - возвращает количество доступных платформ OpenCL. Если num\_platforms равно NULL, этот аргумент игнорируется.

Функция возвращает значение CL\_SUCCESS в случае успешного вызова. В противном случае возвращается одна из следующих ошибок:

* CL\_INVALID\_VALUE если num\_entries равно нулю, а platforms- нет, NULL или если и num\_platforms*,* и платформы равны NULL.
* CL\_OUT\_OF\_HOST\_MEMORY если не удается выделить ресурсы, требуемые реализации OpenCL на хосте.

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);

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 );

* platform - относится к идентификатору платформы, возвращаемому clgetPlatfromIDs() или может быть NULL. Если platform есть NULL, поведение определяется реализацией.
* prm\_name - это константа перечисления, которая идентифицирует запрашиваемую информацию о платформе. Это может быть одно из следующих значений, указанных в таблице [запросов платформы](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#platform-queries-table).
* prm\_value\_size - eказывает размер в байтах памяти, на которую указывает param\_value . Этот размер в байтах должен быть больше или равен размеру возвращаемого типа, указанного в таблице запросов платформы.
* prm\_value - это указатель на ячейку памяти, где будут возвращены соответствующие значения для данного param\_name, как указано в таблице [запросов платформы](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#platform-queries-table). Если param\_value это NULL, она игнорируется.
* prm\_value\_size\_ret - возвращает фактический размер в байтах данных, запрашиваемых параметром param\_name. Если param\_value\_size\_ret есть NULL, он игнорируется.

В случае успеха функция возвращает CL\_SUCCESS. В противном случае возвращается одна из следующих ошибок:

* CL\_INVALID\_PLATFORM - если platform не является действующей.
* CL\_INVALID\_VALUE - если param\_name не является одним из поддерживаемых значений или если размер в байтах, указанный параметром param\_value\_size*,* равен наибольшему размеру возвращаемого типа, как указано в таблице [запросов платформы](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#platform-queries-table) OpenCL , а param\_value не является NULL значением.
* CL\_OUT\_OF\_HOST\_MEMORY - если не удается выделить ресурсы, требуемые реализацией OpenCL на хосте.

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

cl\_int clGetDeviceIDs(

cl\_platform\_id platform ,

cl\_device\_type device\_type ,

cl\_uint num\_entries ,

cl\_device\_id \*devices ,

cl\_uint \*num\_devices );

* platform - относится к идентификатору платформы, возвращаемому  [clGetDeviceIDs](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#clGetPlatformIDs), или может быть NULL. Если platform есть NULL, поведение определяется реализацией.
* device\_type - это битовое поле, которое определяет тип устройства OpenCL. Device\_type может быть использован для запроса конкретных устройств OpenCL или всех устройств OpenCL доступны. Допустимые значения для device\_type указаны в таблице типов устройств.
* num\_entries - это количество записей, которые можно добавить к devices . Если devicesнет , num\_entries должно быть больше нуля.
* devices - возвращает список найденных устройств OpenCL. Эти значения , возвращенные в devices могут быть использованы для идентификации конкретного устройства OpenCL. Если devices есть , этот аргумент игнорируется. Количество возвращаемых устройств OpenCL является минимальным из значения, указанного параметром num\_entries*,* или количества устройств OpenCL, тип которых соответствует device\_type.
* num\_devices - возвращает количество доступных устройств OpenCL, соответствующих device\_type. Если num\_devices равно NULL, этот аргумент игнорируется.

Тип устройства носит чисто информационный характер и не имеет смыслового значения.

Некоторые устройства могут быть нескольких типов. Например, устройство также может быть устройством, или устройство также может быть каким-либо другим, более описательным типом устройства. устройства нельзя комбинировать с устройствами других типов [CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_CPU)\_DEVICE\_TYPE\_CPU[CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_GPU)\_DEVICE\_TYPE\_GPU[CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_ACCELERATOR)\_DEVICE\_TYPE\_ACCELERATOR[CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_CUSTOM)\_DEVICE\_TYPE\_CUSTOM.

Одно устройство на платформе должно быть устройством. Устройство по умолчанию также должно быть более конкретным типом устройства, например или [CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_DEFAULT)\_DEVICE\_TYPE\_DEFAULT[CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_CPU)\_DEVICE\_TYPE\_CPU[CL](https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html#CL_DEVICE_TYPE_GPU)\_DEVICE\_TYPE\_GPU.

В случае успеха функция возвращает CL\_SUCCESS. В противном случае возвращается одна из следующих ошибок

* CL\_INVALID\_PLATFORM - если platform не является действующей платформой.
* CL\_INVALID\_DEVICE\_TYPE - если device\_type не является допустимым значением.
* CL\_INVALID\_VALUE - если num\_entries равно нулю, а devices нет, NULL или если и num\_devices*,* и devicesравны NULL.
* CL\_DEVICE\_NOT\_FOUND - если не найдено ни одного устройства OpenCL, соответствующего device\_type.
* CL\_OUT\_OF\_RESOURCES - если не удается выделить ресурсы, необходимые для реализации OpenCL на устройстве.
* CL\_OUT\_OF\_HOST\_MEMORY - если не удается выделить ресурсы, требуемые реализацией OpenCL на хосте.

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.

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

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 );

* Device - может быть устройством, возвращаемым clGetDeviceIDs,или под-устройствомсозданным clCreateSubDevices. Если devices является под-устройством, будет возвращена конкретная информация о под-устройстве. Информация, которую можно запросить с помощью clGetDeviceInfo, указывается в таблице запросов устройства.
* Param\_name - это константа перечисления, которая идентифицирует запрашиваемую информацию об устройстве. Это может быть одно из следующих значений, указанных в таблице запросов устройства.
* Prm\_value\_size - это указатель на ячейку памяти, где будут возвращены соответствующие значения для заданного param\_name, как указано в таблице запросов устройства. Если param\_value это NULL, она игнорируется.
* Prm\_value - указывает размер в байтах памяти, на которую указывает *param\_value* . Этот размер в байтах должен быть больше или равно размеру возвращаемого типа, указанного в таблице запросов устройства.
* Param\_value\_size\_ret - возвращает фактический размер в байтах данных, запрашиваемых параметром param\_name . Если param\_value\_size\_ret есть NULL, он игнорируется.

Как и остальные функции фреймворка OpenCL, эта функция возвращает CL\_SUCCESS в случае успеха. В противном случае возвращается один из кодов ошибки:

* CL\_INVALID\_DEVICE - если device не является допустимым устройством.
* CL\_INVALID\_VALUE - если param\_name не является одним из поддерживаемых значений или если размер в байтах, указанный параметром param\_value\_size*,* равен наибольшему размеру возвращаемого типа, как указано в таблице запросов устройства, а param\_value не является NULLзначением, или если param\_name является значением, доступным как расширение, а соответствующее расширение не поддерживается устройством.
* CL\_OUT\_OF\_RESOURCES - если не удается выделить ресурсы, необходимые для реализации OpenCL на устройстве.
* CL\_OUT\_OF\_HOST\_MEMORY - если не удается выделить ресурсы, требуемые реализацией OpenCL на хосте.

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

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

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. Контекст - среда исполнения ядер, а также и домен, в котором определены синхронизация и управление памятью. Контекст может создаваться как для одного устройства так и для нескольких или для нескольких очередей команд. 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 .

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 );

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

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);

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.

**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():

cl\_program clCreateProgramWithSource (

cl\_context context ,

cl\_uint count ,

const char \*\*strings ,

const size\_t \*lengths ,

cl\_int \*errcode\_ret );

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.

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 компиляции программы OpenCL, приведенном на листинге (указать номер!!!).

Вначале 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():

cl\_mem clCreateBuffer(

cl\_context context,

cl\_mem\_flags flags,

size\_t size,

void\* host\_ptr,

cl\_int\* errcode\_ret);

* context - это допустимый контекст OpenCL, используемый для создания буферного объекта.
* 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 инициализацию отвечает функция clCreateCommandQueueWithProperties():

cl\_command\_queue clCreateCommandQueueWithProperties(

cl\_context context,

cl\_device\_id device,

const cl\_queue\_properties\* properties,

cl\_int\* errcode\_ret);

* context - должен быть допустимым контекстом OpenCL.
* 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

Модель памяти описывает структуру, содержимое и поведение памяти, предоставленной платформой OpenCL при запуске ядра. Основное назначение данной модели заключается в предоставлении программисту возможности рассуждать о значениях в памяти по мере выполнения основной программы и нескольких экземпляров ядра.

Программа OpenCL содержит в себе контекст, который включает в себя хост, одно или несколько устройств, очереди команд и память, доступную в этом контексте. Программа хоста работает на центральном процессоре в одном или нескольких потоках исполнения, управляемые операционной системой. Ядро же исполняется на одном или нескольких OpenCL устройствах.

Память OpenCL делится на две части:

* Память хоста — память доступна непосредственно самому хосту. Поведение этой памяти не зависит от OpenCL. Объекты памяти перемещаются между хостом и устройствами с помощью OpenCL API или через общий интерфейс виртуальной памяти.
* Память устройства — память, доступная непосредственно самому устройству OpenCL. Доступ к такой памяти имеют только ядра, исполняющиеся на устройствах OpenCL.

Память хоста, её структура и прочее, определяется в рамках определнного вычислительного устройства. Но, память устройства делится на следующие четыре группы:

* глобальная память;
* локальная память;
* константная память;
* приватная память.

К первой категории относятся все область памяти, доступная графическому процессору. К ней имеют доступ все рабочие элементы во всех рабочих группах, работающие на любом устройстве и в любом контексте. Вторая категория характеризует собой только ту область памяти, которая доступна только одной рабочей группы в рамках контекста. Использовать эту область памяти могу все рабочие элементы одной рабочей группы. Третья категория памяти также доступна для использования всем рабочим группам, как и первая, но только для чтения. Четвертая категория памяти используется только один рабочим эдементом.

Глобальная память состоит из **объектов** памяти. Под объектом понимают дескриптор области подсчета ссылок глобальной памяти. Для работы с объектами памяти используется специальный тип данных OpenCL cl\_mem и делится на три различных класса:

* **Буфер —** это объект памяти, хранящийся в виде блока непрерывной памяти и используемый в качестве объекта общего назначения для хранения данных, используемых в программе OpenCL. Типы значений в буфере могут быть любыми встроенными типами (такими как int, float), векторными типами или пользовательскими структурами. Буфером можно манипулировать с помощью указателей так же, как и с любым блоком памяти как в традиционном C.
* **Изображения**  - это объект памяти изображения, содержит одно, два или три трехмерных изображения. Форматы изображения основаны на стандартных графических форматах, используемых в графических приложениях. Изображение - это непрозрачная структура данных, управляемая функциями, определенными в API OpenCL. Чтобы оптимизировать манипуляции с изображениями, хранящимися в текстурных памяти , найденных во многих графических процессорах, ядрам OpenCL традиционно запрещалось как чтение, так и запись одного изображения.
* **Pipe** (канал) — объект памяти, который представляет собой упорядоченную последовательность элементов данных. Канал имеет две конечные точки: конечную точку записи, в которую вставляются элементы данных, и конечную точку чтения, из которой удаляются элементы данных. В любой момент времени только один экземпляр ядра может выполнять запись в канал, и только один экземпляр ядра может читать из канала. Для поддержки шаблона проектирования producer consumer один экземпляр ядра подключается к конечной точке записи (производителю), а другой экземпляр ядра подключается к конечной точке чтения (потребителю).

Объекты памяти выделяются при помощи API хоста. Ведущая программа может предоставить среде выполнения указатель на блок непрерывной памяти для хранения объекта памяти при его создании (CL\_MEM\_USE\_HOST\_PTR). Кроме того, физическая память может управляться средой выполнения OpenCL и не быть непосредственно доступной для основной программы.

Распределение и доступ к объектам памяти в различных областях памяти зависит от хоста и рабочих элементов, работающих на устройстве. Это суммируется в таблице (указать номер!!!) Областей памяти, которая описывает, может ли ядро или хост выделять данные из области памяти, тип выделения (статический во время компиляции или нет). динамический во время выполнения) и тип разрешенного доступа (т. е. Может ли ядро или хост читать и/или записывать данные в область памяти).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Глобальная память | Константная память | Локальная память | Приватная память |
| Хост | Распределение | Динамический | Динамический | Динамический | Нет |
| Доступ | Чтение и запись в буферы и изображения, но не в каналы | Чтение/Запись | Нет | Нет |
| Ядро | Распределение | Статические (переменные области действия программы) | Статические (переменные области действия программы) | Статический для родительского ядра, Динамическая для дочерних ядер | Статическая |
| Доступ | Чтение/Запись | Только для чтения | Чтение/Запись. Нет доступа к памяти дочернего ядра. | Чтение/Запись |

Таблица Областей памяти показывает различные области памяти в OpenCL и то, как объекты памяти выделяются и доступны хосту и исполняющему экземпляру ядра. Для ядер мы различаем поведение локальной памяти для родительского ядра и его дочерних ядер.

После выделения объект памяти становится доступным для экземпляров ядра , работающих на одном или нескольких устройствах. Помимо общей виртуальной памяти, существует три основных способа управления содержимым буферов между хостом и устройствами.

* **Команды чтения/записи/заполнения** - данные, связанные с объектом памяти , явно считываются и записываются между узлом и глобальными областями памяти с помощью команд, помещенных в очередь команд OpenCL.
* **Команды Map/Unmap** - данные из объекта памяти отображаются в непрерывный блок памяти, доступ к которому осуществляется с помощью указателя host. Хост-программа ставит в очередь команду map на блоке объекта памяти , прежде чем он сможет безопасно манипулировать хост-программой. Когда ведущая программа завершает работу с блоком памяти, ведущая программа ставит в очередь команду unmap, чтобы позволить экземпляру ядра безопасно читать и/или записывать буфер.
* **Копирование команд** - данные, связанные с объектом памяти, копируются между двумя буферами, каждый из которых может находиться либо на хосте, либо на устройстве.

Команды Read/Write/Map могут быть блокирующими или неблокирующими операциями. Вызов функции OpenCL для блокирующей передачи памяти возвращается после завершения команды (передачи памяти). На этом этапе связанные ресурсы памяти на хосте могут быть безопасно использованы повторно, и последующие операции на хосте гарантируют, что передача уже завершена. Для неблокирующей передачи памяти вызов функции OpenCL возвращается , как только команда поставлена в очередь.

Объекты памяти привязаны к контексту и, следовательно, могут появляться в нескольких экземплярах ядра, работающих на нескольких физических устройствах. Платформа OpenCL должна поддерживать широкий спектр аппаратных платформ , включая системы, которые не поддерживают единое общее адресное пространство в аппаратном обеспечении; следовательно, способы совместного использования объектов памяти между экземплярами ядра ограничены. Основной принцип заключается в том, что допускается несколько операций чтения с объектами памяти из нескольких экземпляров ядра, которые перекрываются во времени, но смешивание перекрывающихся операций чтения и записи в одни и те же объекты памяти из разных экземпляров ядра. экземпляры ядра разрешены только при использовании мелкозернистой синхронизации с Общая виртуальная Память.

Когда глобальная память управляется несколькими экземплярами ядра, работающими на нескольких устройствах, система выполнения OpenCL должна управлять связью объектов памяти с данным устройством. В большинстве случаев среда выполнения OpenCL неявно связывает объект памяти с устройством. Экземпляр ядра естественным образом связан с очередью команд, в которую было отправлено ядро. Поскольку очередь команд может обращаться только к одному устройству, очередь однозначно определяет, какое устройство задействовано с любым заданным экземпляром ядра; таким образом , определяя четкую связь между объектами памяти, экземплярами ядра и устройства. Программисты могут предвидеть эти ассоциации в своих программах и явно управлять ассоциацией объектов памяти с устройствами, чтобы повысить производительность.

В данном разделе были рассмотрены общие вопросы устройства памяти OpenCL, за более подробным описание стоит обратиться (указать источник!!!).

## 3.4.4. Модель программирования

Разработка программ под устройства OpenCL осуществляется на специальном языке программирования OpenCL C. По мере совершенствования фрейморвка OpenCL язык приобретал новые функциональные возможности и синтаксические конструкции. Для более полного и глубоко изучения данного вопроса стоит обратиться к источнику (указать источник!!!). В данном же разделе рассмотрим общие концепции, лежащие о основе языка программирования OpenCL C.

Синтаксис языка во многом схож со стандартом языка программирования C99. В него входят те же условные операторы и арифметические операторы и операторы присваивания, типы данных и прочее.

Для начала рассмотрим типы данных, присущие языку программирования.

Условной типы данных языка программирования OpenCL C можно разделить на две категории: скалярные (базовые) типы данных и векторные типы данных. Перечислим все типы данных в двух таблицах, для каждой из категории.

|  |  |  |
| --- | --- | --- |
| Тип | Описание | OpenCL API |
| bool | Условный тип данных, который является либо истинным (true), либо ложным(false). Значение true расширяется до целочисленной константы 1 и значения false расширяется до целочисленной константы 0. | n/a |
| char | Знаковое двоичное дополнение 8-битного целого числа. | cl\_char |
| unsigned char, uchar | 8-битное целое число без знака. | cl\_uchar |
| short | Знаковое двоичное дополнение 16-битного целого числа. | cl\_short |
| unsigned short, ushort | 16-битное целое число без знака | cl\_ushort |
| int | 32-битное целое число дополнения двойки со знаком. | cl\_int |
| unsigned int, uint | 32-разрядное целое число без знака. | cl\_uint |
| long | Знак два дополняет 64-битное целое число. | cl\_long |
| unsigned long,  ulong | 64-разрядное целое число без знака. | cl\_ulong |
| float | 32-битная плавающая точка. Тип floatданных должен соответствовать формату хранения с одной точностью IEEE 754. | cl\_float |
| double | 64-битная плавающая точка. Тип doubleданных должен соответствовать формату хранения двойной точности IEEE 754. | cl\_double |
| half | 16-битная плавающая точка. Тип halfданных должен соответствовать формату хранения половинной точности IEEE 754-2008. | cl\_half |
| size\_t | Беззнаковый целочисленный тип результата sizeof оператора. | n/a |
| ptrdiff\_t | Знаковый целочисленный тип, который является результатом вычитания двух указателей. | n/a |
| intprt\_t | Знаковый целочисленный тип со свойством, что любой допустимый указатель на void может быть преобразован в этот тип , а затем преобразован обратно в указатель на void, и результат будет сравниваться равным исходному указателю. | n/a |
| uintptr\_t | Целочисленный тип без знака со свойством, что любой допустимый указатель на void может быть преобразован в этот тип , а затем преобразован обратно в указатель на void, и результат будет сравниваться равным исходному указателю | n/a |
| void | Тип содержит пустой набор значений; это неполный тип, который не может быть завершен. | void |

Теперь рассмотрим векторные типы данных.

|  |  |  |
| --- | --- | --- |
| Тип | Описание | OpenCL API |
| char*n* | Вектор из n 8-битных знаковых двух дополняющих целочисленных значений. | cl\_char*n* |
| uchar*n* | Вектор из n 8-битных целочисленных значений без знака. | cl\_uchar*n* |
| short*n* | Вектор из n 16-битных знаковых двух дополняющих целочисленных значений. | cl\_short*n* |
| ushort*n* | Вектор из n 16-битных целочисленных значений без знака. | cl\_ushort*n* |
| int*n* | Вектор из n 32-битных знаковых двух дополняющих целочисленных значений. | cl\_int*n* |
| uint*n* | Вектор из n 32-битных целочисленных значений без знака. | cl\_uint*n* |
| long*n* | Вектор из n 64-битных знаковых двух дополняющих целочисленных значений. | cl\_long*n* |
| ulong*n* | Вектор из n 64-битных целочисленных значений без знака. | cl\_ulong*n* |
| float*n* | Вектор из n 32-битных значений с плавающей запятой. | cl\_float*n* |
| double*n* | Вектор из n 64-битных значений с плавающей запятой. | cl\_double*n* |

В следующей таблице представлены дополнительные типы данных

## 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.

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.

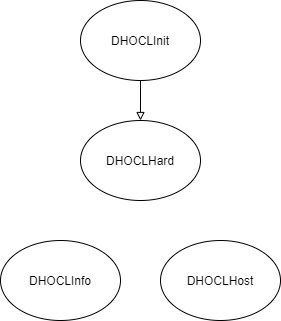
## 4.2. Использование фреймворка OpenCL

Условно взаимодействие с фреймворком OpenCL происходит в два этапа. На первом этапе происходит инициализация платформ и устройств OpenCL. На втором этапе происходит создание контекста, программы, очереди команд, ядер и их исполнение. Рассмотрим реализацию этих этапов в целевом программном обеспечении.

## 4.2.1. Инициализация платформ и устройств OpenCL

Для решения данной задачи балы разработана подсистема классов, которая производит выборку и загрузку дополнительной информации о всех существующих платформах и устройствах OpenCL на компьютере.

На рисунке (указать номер!!!) представлена диаграмма классов, которые осуществляют инициализация платформ и устройств OpenCL, а также определяют о них необходимую информацию. Рассмотрим каждый из этих классов подробнее.

Рис. . Диаграмма классов инициализации платформ и устройств OpenCL.

На листинге (указать номер!!!) представлено определение класса DHOCLInit. Данный класс отвечает за инициализацию поля m\_Hardware, которое является объектом класса std::vector<DHOCLHard>.

За инициализацию данного поля отвечает метод Init, реализация которого представлена на листинге (указать номер!!!). В данном методе происходит вызов стандартных функций OpenCL API для инициализации платформ, обсуждаемых в разделе (указать номер!!!). После определения платформ происходит инициализация устройств, посредством создания объекта класса DHOCLHard. Затем полученный объект помещается в поле m\_Hardware.

В случае успешного вызова функции возвращается значение true. В противном случае — false.

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;

};

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)

throw dholo::exp::DHGPGPUExp(err);

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

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

if (err != CL\_SUCCESS)

throw dholo::exp::DHGPGPUExp(err);

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

{

DHOCLHard hard(platfroms[i]);

m\_Hardware.push\_back(hard);

}

return true;

}

Остальные методы класса играют вспомогательную роль. Метод GetSize, возвращает размер вектора m\_Hardware. Метод GetHardware и оператор [] возвращают конкретный объект из поля m\_Hardware.

Поле m\_Hardware состоит из объектов класса DHOCLHard. Этот класс отвечает за хранение платформ и всех устройств OpenCL, находящихся под управлением, в одном объекте. Такой подход позволяет обращаться к любой платформе и к любому устройству OpenCL. Рассмотрим определение данного класса на листинге (указать номер!!!).

Данный класс состоит из полей m\_Platform, типа cl\_platform\_id; поля m\_Devices, типа cl\_device\_id\* и поля m\_Size, типа cl\_uint. Первое поле хранит в себе указатель на платформу OpenCL, передаваемую в класс через конструктор с параметром. Второе поле хранит в себе все устройства OpenCL, связанные с данной платформой. Третье поле хранит в себе количество устройств OpenCL.

Инициализация данных полей происходит в конструкторе класса с параметром cl\_platform\_id. Его реализация представлена на листинге (указать номер!!!). При вызове конструктора происходит присвоение поля m\_Platform значения, переданного в качестве аргумента вызова метода. Затем происходит инициализация остальных полей посредством вызова стандартной функции clGetDeviceIDs(), обсуждаемой в разделе (указать номер!!!).

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;

};

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

{

m\_Platform = platform;

m\_Size = 0;

m\_Devices = nullptr;

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

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

throw dholo::exp::DHGPGPUExp(err);

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)

throw dholo::exp::DHGPGPUExp(err);

}

В случае ошибки будет выбрасываться исключение типа DHGPGPUExp.

Также у данного класса имеется деструткор, который вызывает функцию уничтожения поля m\_Devices при помощи вызова функции clReleaseDevice().

Остальные методы класса играют вспомогательную роль, рассмотрим их синтаксис и назначение в таблице (указать номер!!!).

|  |  |
| --- | --- |
| Метод | Назначение |
| GetCountDevices() | Возвращает количество устройств OpenCL |
| GetDevice(unsigned int index) | Возвращает устройств OpenCL по индексу |
| GetPlatform() | Возвращает платформу OpenCL |
| operator[](int index) | Возвращает устройств OpenCL по индексу |

Остальные классы из файла DHOCLInit носят вспомогательный характер. В листинге (указать номер!!!) представлено определение класса DHOCLHost.

class DHOCLHost

{

public:

DHOCLHost();

DHOCLHost(cl\_platform\_id, cl\_device\_id);

~DHOCLHost();

void SetPlatform(cl\_platform\_id);

void SetDevice(cl\_device\_id);

cl\_platform\_id GetPlatform() const;

cl\_device\_id GetDevice() const;

private:

cl\_platform\_id m\_Platform;

cl\_device\_id m\_Device;

};

Данный класс хранит в себе конкретные значения поля и устройства OpenCL для простой передачи данных. Класс состоит из двух полей m\_Platform типа cl\_platform\_id и поля m\_Device типа cl\_device\_id. За инициализацию этих полей отвечает конструктор класс с пераметрами и методы SetDevice SetPlatform. Первый инициализирует поле m\_Device, второе инициализирует m\_Platform. Остальные два метода GetDevice и GetPlatform возвращают устройство и платформу OpenCL.

Другим вспомогательным классом является DHOCLInfo, определение которого представлено на листинге (указать номер!!!).

Все методы данного класса являются статическими и в основе своей вызывают стандартные функции OpenCL API для определения информации об платформе и устройстве OpenCL. Рассмотрим назначение каждого из методов в таблице (указать номер!!!).

class DHOCLInfo

{

public:

DHOCLInfo() = default;

~DHOCLInfo() = default;

static std::wstring GetPlatformName(cl\_platform\_id pl);

static std::wstring GetPlatformVendor(cl\_platform\_id pl);

static std::wstring GetPlatformVersion(cl\_platform\_id pl);

static std::wstring GetPlatformExtensions(cl\_platform\_id pl);

static std::wstring GetDeviceName(cl\_device\_id dev);

static std::wstring GetDeviceVendor(cl\_device\_id dev);

static std::wstring GetDeviceVersion(cl\_device\_id dev);

static std::wstring GetDeviceExtensions(cl\_device\_id dev);

static std::wstring GetDeviceType(cl\_device\_id dev);

static std::wstring GetDeviceFrequency(cl\_device\_id dev);

static std::wstring GetDeviceGlobalSize(cl\_device\_id dev);

static std::wstring GetDeviceLocalSize(cl\_device\_id dev);

static std::wstring GetPlatformInfo(cl\_platform\_id pl, cl\_platform\_info inf);

static std::wstring GetDeviceInfo(cl\_device\_id dev, cl\_device\_info inf);

};

|  |  |
| --- | --- |
| Метод | Назначение |
| GetPlatformName(cl\_platform\_id pl) | Возвращает имя платформы OpenCL |
| GetPlatformVendor(cl\_platform\_id pl) | Возвращает производителя платофрмы OpenCL |
| GetPlatformVersion(cl\_platform\_id pl) | Возвращает версию платформы OpenCL |
| GetPlatformExtensions(cl\_platform\_id pl) | Возвращает расширения платформы OpenCL |
| GetDeviceName(cl\_device\_id dev) | Возвращает имя устройства OpenCL |
| GetDeviceVendor(cl\_device\_id dev) | Возвращает производителя устройства OpenCL |
| GetDeviceVersion(cl\_device\_id dev) | Возвращает версию устройства OpenCL |
| GetDeviceExtensions(cl\_device\_id dev) | Возвращает расширения устройства OpenCL |
| GetDeviceType(cl\_device\_id dev) | Возвращает тип устройства OpenCL |
| GetDeviceFrequency(cl\_device\_id dev) | Возвращает тактовую частоту устройства OpenCL |
| GetDeviceGlobalSize(cl\_device\_id dev) | Возвращает размер глобальной группы доступной на устройстве OpenCL |
| GetDeviceLocalSize(cl\_device\_id dev) | Возвращает размер локальной группы на устройстве OpenCL |
| GetPlatformInfo(cl\_platform\_id pl, cl\_platform\_info inf) | Вспомогательный метод для возвращения конкретной информации о платформе OpenCL |
| GetDeviceInfo(cl\_device\_id dev, cl\_device\_info inf) | Вспомогательный метод для возвращения конкретной информации об устройстве OpenCL |

## 4.2.2. Инициализация структур данных OpenCL для исполнения программ

Для решения данной задачи была разработана иерархия классов. Каждый класс реализует собой определенной функционал приложения для расчетов на OpenCL устройствах.

## 4.3. Разработка графического интерфейса пользователя

Для разработки оконного интерфейса Windows используется библиотека MFC. Представим диаграмму классов, реализующих графический интрейфес пользователя на рисунке (указать номер!!!).

Диаграмма состоит из четырех частей. На первой части размещается иерархия классов, необходимых для запуска приложения и создания окна. Эти классы предоставляются самой библиотекой MFC. Класс DHApp является наследником класса CWinApp. Переопределение методов реализуется самой библиотекой MFC, дополнительных изменений не производилось. На листинге (указать номер!!!) приведено определение данного класса.

## 4.3.1. Разработка основы приложения

Вторым классом в иерархии является класс DHMainFrm, которыя является наследником класса CFrameWndEx. Его определение приведено на листинге (указать номер!!!). Основным назначением данного класса является перехват системных сообщений и выполнении необходимых действий. В таблице (указать номер!!!) приведены методы класса и их назначение, которые осуществляют основной функционал приложения.

|  |  |
| --- | --- |
| Метод | Назначение |
| OnMenu | Вызов контекстного меню |
| OnDeleteImage | Удаление изображения из ленты |
| OnAddImage | Добавление изображения в ленту |
| OnLoadIntoGp | Загрузка изображения в видеопамять |
| OnPSITransform | Выполнение метода пошагового фазового сдвига целевого изображения в видеопамяти графического ускорителя |
| OnGetAmplitude | Определение амплитуды из значений целевого изображения в видеопамяти графического ускорителя |
| OnGetPhase | Определение фазы из значений целевого изображения в видеопамяти графического ускорителя |
| OnGetRe | Определение целой части из значений целевого изображения в видеопамяти графического ускорителя |
| OnGetIm | Определение мнимой части из значений целевого изображения в видеопамяти графического ускорителя |
| OnGenSin | Генерация синуса |
| OnSaveImg | Сохранение изображения на диск |
| OnSaveAndAddImg | Сохранение изображения на диск и добавление его в ленту изображений |

Остальные методы класса носят общий характер и предоставляются самой библиотекой MFC. Они играть вспомогательную роль и призваны осуществлять необходимые операции по работе с операционной системой Windows. Из список и назначение приведены в таблице (указать номер!!!).

|  |  |
| --- | --- |
| Метод | Назначение |
| PreCreateWindow | Вызывается перед вызовом метода OnCreate. Необходим для инициализации или загрузки необходимых данных для работы окна |
| OnCmdMsg | Данный метод наследуется от класса CcmdTarget. Осуществляет маршрутизацию и отправляет командные сообщения. |
| LoadFrame | Данный метод осуществляет динамическое создание фрейма из файла ресурсов |
| AssertValid | Данный метод наследуется от класса Cobejct. Осуществляет проверку целостности объекта |
| Dump | Данный метод наследуется от класса CObejct. Создает дамп процесса в случае возникновения исключительной ситуации |
| OnCreate | Используется при создании объекта окна |
| OnSetFocus | Данный метод вызывается когда объект CWnd получает фокус ввода |
| OnToolbarCreateNew | Используется для загрузки toolbar из файла ресурсов и отображения его во фрейме |

Следующим в очереди является класс DHWnd. Этот класс осуществляет вывод изображения в главном окне приложения и выполняет необходимые операции преобразования над голограммой(-ами). Определение данного класса приведено на листинге (указать номер!!).

Перечислим методы данного класса и их назначение в таблице (указать номер!!!).

|  |  |
| --- | --- |
| Метод | Назначение |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Данный класс является мостом между графическим интерйесом пользователя и графическим процессором. Именно через этот класс выполняется загрузка изображения в видеопамять и выполнение программ для графического процессора. Загрузка и выполнение преобразования над изображение происходит в методе OnPaint. В нем вызываются методы класса из предыдущего пункта.

## 4.3.2. Разработка системы исключений

Исключения является механизмом языка программирования C++ для обозначения возникновения ошибки или исключительной ситуации. Например невозможность выделить необходимый объем памяти, неправильное имя файла и т.д. Для разрабатываемого приложения создана система исключений из двух частей. Первая часть необходима для вывода ошибок от операционной системы. Этой часть занимается класс DHAppExp, его определение представлено на листинге (указать номер!!!).

Класс DHAppExp является наследником класс std::exception для возможности генерации исключений данного типа. Класс состоит ровно из одного поля m\_ErrorMessage типа CString. Класс Cstring является часть библиотеки MFC, используется для работы со строками в различных кодировках.

Помимо поля класс DHAppExp включает в себя два конструктора с параметрами. В каждый из них передается сообщение об ошибке. Разницей между ними является то, что первый принимает сроку в кодировки UTF-8,а второй в UTF-16. Другим важным методом является ShowError, который вызывается в блоке catch. Данный метод занимается выводом ошибки в виде окна сообщений.

class DHAppExp : public std::exception

{

public:

DHAppExp() = default;

DHAppExp(const std::string& mess);

DHAppExp(const CString& mess);

~DHAppExp() = default;

void ShowError() const;

protected:

CString m\_ErrorMessage;

};

Вторая часть системы исключений занимается обработкой ошибок от OpenCL API. Используется в классах из предыдущего пункта. Данный класс DHGPGPUExp наследуется от класса DHAppExp, его определение представлено на листинге (указать номер!!!).

В этом классе присутствует два конструктора с параметрами. Первый принимает код ошибки в стиле OpenCL API. При вызове этого конструктора осуществляется обращение к файлу ресурсов, в котором хранится STRINGTABLE с заранее определенными кодами ошибок и их описании. Второй конструктором работает аналогично первому конструктору. Разница состоит в том, что к исходному сообщению об ошибке из файла ресурсов добавляется дополнительная информация об ошибке из аргумента mess. Этот конструктор используется в случае возникновения ошибки компилятора OpenCL, которое уточняет ошибку компилятора.

class DHGPGPUExp : public DHAppExp

{

public:

DHGPGPUExp(cl\_int error\_code);

DHGPGPUExp(cl\_int error\_code, const std::string &mess);

~DHGPGPUExp();

};

## 4.3.3. Разработка ленты изображений

Третьей часть графического интерфейса пользователя является лента изображений. Ее назначение заключается в представлении и использовании изображений в удобном виде. Данная лента работает за счет трех классов DHImgExpr, DHImgList и DHImgLoader.

На листинге (указать номер!!!) представлено определение класса DHImgList. Данный класс наследуется от класса CCtrlList из библиотеки MFC. Данный класс предназначен для создания элемента управления список.

Вторым по значимости классом является DHImgExpr, назначение которого заключается в загрузке и удалении изображений. Его определение представлено на листинге (указать номер!!!).

class DHImgList : public CListCtrl

{

public:

DHImgList() noexcept;

virtual ~DHImgList();

protected:

virtual BOOL OnNotify(WPARAM wParam, LPARAM lParam, LRESULT\* pResult);

DECLARE\_MESSAGE\_MAP()

};

Данный класс является наследником класса CDockablePane из библиотеки MFC. Данный класс используется в классе DHMainFrm для отображения ленты в главном окне приложения. Данный элемент управления способен перемещаться по главному окну и выйти за его рамки и стать дочерним по отношению к главному. Перечислим методы данного класса и их назначение в таблице (указать номер!!!).

|  |  |
| --- | --- |
| Метод | Назначение |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Следующим классом из списка является класс DHImgLoader, его определение представлено на листинге (указать номер!!!).

Данный класс осуществляет загрузку изображения с диска по пути к файлу. Данный класс работает при помощи библиотеки stb\_image. Помимо этого класс предоставляет массив пикселей типа float, ширину и высоту изображения и количество каналов изображения. Методы и их назначение представлено в таблице (указать номер!!!).

class DHImgLoader

{

public:

DHImgLoader();

~DHImgLoader();

bool Load(const CStringA& imgPath);

int GetWidth() const;

int GetHeight() const;

int GetChannels() const;

float\* GetPixelsData() const;

void GenerateImage(int width, int height, int channels);

void SetPixels(int width, int height, int channles, float \*pxls);

private:

float\* m\_PixelsData;

int m\_width;

int m\_height;

int m\_nrChannels;

};

|  |  |
| --- | --- |
| Метод | Назначение |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

В совокупности все эти классы осуществляют функционал ленты изображений и загрузки необходимо изображения в приложение.

## 4.3.4. Разработка вспомогательного функционала

Остальные классы носят вспомогательный характер. Например класс DHToolBar осуществляет функционал главного элемента управления toolbar. Этот класс используется для создания трех toolbar элементов управления в приложении. К этому числу относится главный toolbar приложения, который располагается над лентой изображений. Второй toolbar располагается в ленте изображений и осуществляет загрузку, удаление изображения и PSI преобразование. Третий toolbar располагается рядом с первым и предоставляет функционал по преобразования изображения в графической ускорителе. Определение данного класса приведено в листинге (указать номер!!!).

Данный класс является наследником класса CMFCToolBar библиотеки MFC.

class DHToolBar : public CMFCToolBar

{

virtual void OnUpdateCmdUI(CFrameWnd\* /\*pTarget\*/, BOOL bDisableIfNoHndler)

{

CMFCToolBar::OnUpdateCmdUI((CFrameWnd\*)GetOwner(), bDisableIfNoHndler);

}

virtual BOOL AllowShowOnList() const { return FALSE; }

};

Следующим вспомогательный классом является DHGLGSinus, его назначение заключается в выводе диалога для генерации синусоидальной волны. В данном диалоге предоставляется возможность выбора амплитуды, фазы и количества периодов. Определение данного класса представлено на листинге (указать номер!!!).

Методы и их назначение приведено в таблице (указать номер!!!).

class DHGENSinus : public CDialogEx

{

DECLARE\_DYNAMIC(DHGENSinus)

public:

DHGENSinus(CWnd\* pParent = nullptr);

virtual ~DHGENSinus();

virtual void DoDataExchange(CDataExchange\* pDX);

float GetAmpl() const;

float GetPhase() const;

int GetWidth() const;

int GetHeight() const;

int GetChannels() const;

int GetT() const;

protected:

afx\_msg BOOL OnInitDialog();

afx\_msg void OnBnClickedCheckPhaseRandom();

afx\_msg void OnOK();

DECLARE\_MESSAGE\_MAP()

private:

float m\_Ampl;

float m\_Phase;

int m\_Width;

int m\_Height;

int m\_Channels;

int m\_T;

};

|  |  |
| --- | --- |
| Метод | Назначение |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Последним вспомогательным классом является DHOCLTransDlg, который предоставляет возможность настройки метода пошагового фазового сдвига. Предоставляется возможность выбора устройства и платформы OpenCL и дополнительная информация об них, а также настройка фаз для четырех изображений. Определение класса предоставлено на листинге (указать номер!!!).

Методы и их назначение предоставлено в таблице (указать номер!!!).

class DHOCLTransDlg : public CDialogEx

{

DECLARE\_DYNAMIC(DHOCLTransDlg)

public:

DHOCLTransDlg(CWnd\* pParent = nullptr);

virtual ~DHOCLTransDlg();

virtual void DoDataExchange(CDataExchange\* pDX);

afx\_msg BOOL OnInitDialog();

afx\_msg void OnOK();

afx\_msg void OnCancel();

afx\_msg void OnCbnSelchangeComboPlatformName();

afx\_msg void OnCbnSelchangeComboDeviceName();

afx\_msg void OnBnClickedCheckPhaseRandom();

afx\_msg void OnBnClickedCheckPhaseRandom2();

afx\_msg void OnBnClickedCheckPhaseRandom3();

afx\_msg void OnBnClickedCheckPhaseRandom4();

void FillInComboPlatforms();

void FillInPlatformPanel();

void FillInComboDevice();

void FillInDevicePanel();

void FillInPSIPanel();

void FillInComboPhase(int code);

void FillEdit(int id, const std::wstring& str);

float GetPhase1() const;

float GetPhase2() const;

float GetPhase3() const;

float GetPhase4() const;

dholo::gpgpu::DHOCLHost GetHost() const;

void CheckRandomPahesBox(int check, int combo, int edit);

DECLARE\_MESSAGE\_MAP()

private:

gpgpu::DHOCLInit m\_Init;

gpgpu::DHOCLHost m\_Host;

float m\_Phase1;

float m\_Phase2;

float m\_Phase3;

float m\_Phase4;

float m\_B;

};

|  |  |
| --- | --- |
| Метод | Назначение |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Помимо вспомогательных классов существуют и вспомогательные функции, например GetNumberFromEdit, определение и реализация которой предоставлена на листинге (указать номер!!!).

Данный метод используется в классах диалогов для заполнения элементов управления типа edit.

template<typename T = int>

inline T GetNumberFromEdit(CEdit\* edit)

{

CString strNumber;

edit->GetWindowTextW(strNumber);

return static\_cast<T>(std::atoi(static\_cast<CStringA>(strNumber)));

}

Помимо этой функции существует еще один вспомогательный метод класса DHOCLTransDlg для инициализации элементов управления диалога, определение и реализация этого метода приведены на листинге (указать номер!!!).

template<typename T = int>

inline T GetNumberFromControl(int checkBoxID, int editID, int comboID)

{

auto checkBoxMyPhase = static\_cast<CButton\*>(this->GetDlgItem(checkBoxID));

int checked = checkBoxMyPhase->GetCheck();

if (checked == BST\_CHECKED)

{

auto phase = static\_cast<CEdit\*>(this->GetDlgItem(editID));

return GetNumberFromEdit<T>(phase);

}

else

{

auto phase = static\_cast<CComboBox\*>(this->GetDlgItem(comboID));

int numerator = phase->GetCurSel();

return numerator \* CL\_M\_PI\_2;

}

}

# 5. ТЕСТИРОВАНИЕ

# ЗАКЛЮЧЕНИЕ

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК

# ПРИЛОЖЕНИЕ