МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«Санкт–Петербургский государственный университет  
аэрокосмического приборостроения»

Кафедра №43 «Компьютерных технологий и программной инженерии»

ОТЧЁТ ПО ПРАКТИКЕ

ЗАЩИЩЁН С ОЦЕНКОЙ

Руководитель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ст. преподаватель |  |  |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

ОТЧЁТ ПО ПРАКТИКЕ

|  |  |  |
| --- | --- | --- |
| вид практики | производственная | |
| тип практики | технологическая (проектно-технологическая) | |
| на тему индивидуального задания | |  |
|  | | | |
|  | | | |

|  |  |
| --- | --- |
| выполнен |  |
| фамилия, имя, отчество обучающегося в творительном падеже | |

|  |  |  |  |
| --- | --- | --- | --- |
| по направлению подготовки | 09.03.04 |  | Программная инженерия |
|  | код |  | наименование направления |
|  | | | |
| наименование направления | | | |
| направленности | 01 |  | Проектирование программных систем |
|  | код |  | наименование направленности |
| систем | | | |
| наименование направленности | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Обучающийся группы № | Z9431 |  |  |  | Д.И. Андреев |
|  | номер |  | подпись, дата |  | инициалы, фамилия |

Санкт–Петербург 2021

Оглавление

[1. Цель работы 3](#_Toc70018215)

[2. Исходные данные 4](#_Toc70018216)

[3. Теоретический раздел 5](#_Toc70018217)

[4. Практический раздел 10](#_Toc70018218)

[5. Результаты 15](#_Toc70018219)

[6. Выводы 16](#_Toc70018220)

[Приложение 1 17](#_Toc70018221)

[Список литературы 25](#_Toc70018222)

1. Цель работы

Целью производственной практики является реализация алгоритмов кодирования и декодирования кадров, захваченных с монитора машины для их потоковой передачи по протоколу UDP в виде потокового видео.

1. Исходные данные

Исходными данными являются:

* QImage – объект класа, содержащий данные о захваченном с использованием Qt Framework изображении произвольного размера;
* Программа, реализующая передачу данных с использованием UDP протокола;
* Алгоритм пространственного предсказания в соответствии со стандартом HEVC H.265

1. Теоретический раздел

При передаче потокового видео решающим фактором является размер каждого передаваемого кадра. Чем больше его размер, тем больше предъявляются требования к пропускной способности канала передачи для уменьшения задержки видео. Например, для видео разрешением 480x240 скоростью 30 кадров в секунду и 24 битов на пиксел потребуется пропускная способность канала передачи 480х240х24х30 = 82 944 000 бит/c или 82,944 Мбит/с.

Для уменьшения размера передаваемой информации используются различные методы сжатия, в частности уменьшение пространственной и временной избыточности [1]. Оба способа сжатия заключаются в том, чтобы передавать клиенту только ту информацию об изображении, которой у него еще нет.

**Временное уменьшение избыточности.**

Так как в любом видео разница между кадрами незначительная, основная часть информации, которую несет в себе текущий кадр уже есть у клиента. Разница между текущим и предыдущим кадром состоит только в объектах, изменивших свое положение. Временное уменьшение избыточности или межкадровое предсказание заключается в том, чтобы получать на основе этих изображений только движущиеся объекты.

Для реализации данного вида предсказания необходимы как минимум 2 референсных кадра - I-кадр (intra-frame) и B-кадр (Bi-predictive-frame), и кодируемый кадр P-кадр (predicted-frame).

I-кадр является опорным. На его основе делаются предсказания о движущихся объектах. Первый кадр обычно является I-кадром, однако при смене сцен I-кадр может быть обновлен. P-кадр является прогнозируемым кадром, для которого осуществляется предсказание на основе текущего I-кадра. B-кадр – будущий кадр, на основе которого также осуществляется предсказание. Так как на основании B-кадров и I-кадров должно происходить кодирование P-кадра, все референсные I- и В-кадры должны быть сохранены в памяти машины до смены сцены, когда происходит смена I-кадра [2]. B-кадры не используются только при прямом предсказании, когда предсказание осуществляется только на основе уже отрисованных кадров.

Перед началом кодирования каждый кадр должен быть поделен квадратные блоки (макроблоки в HEVC [5]), которые могут быть также разделены на 4 квадратных блока. В результате такого деления образуется квадродерево, что позволяет получить большую точность при кодировании.

Суть межкадрового предсказания заключается в том, чтобы найти разность между текущим кодируемым и референсным изображением, получить векторы движения блоков, полученного квадродерева и отправить клиенту только ненулевые вектора движения данных блоков. При этом дополнительно должна быть передана информация о положении блока в квадродереве и референсных кадрах на основе которых было построено предсказание. В HEVC эта информация создается на последнем этапе кодирования при [контекстно-адаптивном двоичном арифметическом кодировани](https://www.elecard.com/ru/page/article_context_adaptive_binary_arithmetic_coding_5)и.

При межкадровом кодировании видеопоследовательности порядок кодирования отличается от порядка воспроизведения [3]. На рис. 1 показан порядок отображения, при котором кадр отображается с использование двух B-кадров. Как можно увидеть, из всех кадров, подлежащих отображению, первым отображается I-кадр. Два B-кадра B1 и B2 отображаются вслед за исходным I-кадром. P-кадр P3 далее опять два B-кадра B4 и B5 и т.д. На рис. 2 можно увидеть последовательность кодирования, при которой каждый кадр отображают с использованием двух B-кадров. Как можно увидеть, если кодируется исходный I-кадр, то Р-кадр Р3 будет кодироваться перед двумя B-кадрами В1 и В2, которые отображаются до Р-кадра Р3. Затем последовательно кодируют кадры Р6, В4, В5, Р9, В7, Р12, В10 и В11.

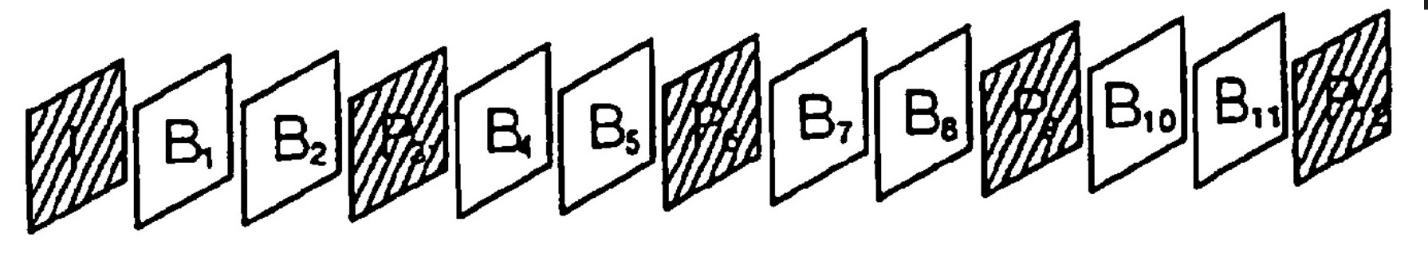


Рис. 1 Последовательность отображения кадров при межкадровом кодировании

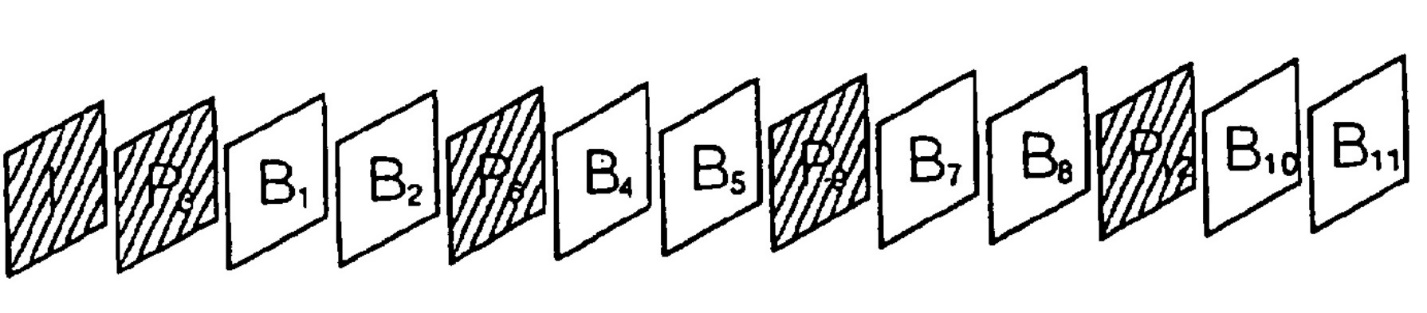


Рис. 2 Последовательность кодирования кадров при межкадровом кодировании

**Пространственное уменьшение избыточности.**

Любой кадр содержит участки с одинаковой информацией о цветах в пределах блока. Если устранить данную избыточность можно отправлять клиенту информацию только об изменении цвета на изображении.

Данный способ кодирования состоит в том, чтобы на основе референсных пикселов получить разностный сигнал между текущим блоком и его усредненным вариантом. Перед началом кодирования, так же, как и для временного предсказания строится квадродерево. Разбиение видеокадра на блоки производится адаптивно, то есть подстраивается под характер изображения. Прежде всего изображение слева-направо и сверху-вниз разбивается на одинаковые квадратные блоки, назывемые LCU (от англ. Largest Coding Unit). Размер LCU является параметром настройки кодирующей системы и задается перед кодированием. Этот размер может принимать значения 8x8, 16x16, 32x32 и 64x64. Каждый блок LCU может быть разбит на 4 квадратных подблока CU, каждый из которых, в свою очередь, может быть также разбит. Таким образом, LCU является корнем квадродерева. Минимальный размер CU или глубина квадродерева также является задаваемым параметром настройки кодирующей системы и может принимать значения 8x8, 16x16, 32x32, 64x64. Пространственное предсказание в HEVC выполняется для блоков квадратной формы, называемых PU (от англ. Prediction Unit). Размер PU совпадает с размером CU с двумя исключениями. Во-первых, размер PU не может превышать 32x32. Таким образом, CU с размером 64x64 содержит внутри себя 4 PU с размерами 32x32. Во-вторых, CU нижнего уровня квадродерева, имеющие минимально разрешенный размер, также могут быть разбиты на 4 квадратных PU с вдвое меньшими размерами. В результате набор разрешенных в HEVC размеров PU состоит из следующих значений: 4x4, 8x8, 16x16, 32x32 [5]. Пример разбиения изображения на блоки CU и PU приведен на рис. 3. Более толстой сплошной линией выделены границы LCU. Границы подблоков CU отмечены тонкой сплошной линией. Пунктиром обозначены границы PU в тех случаях, когда CU содержит 4 PU. Здесь же цифрами внутри блоков указан порядок перебора PU при кодировании.

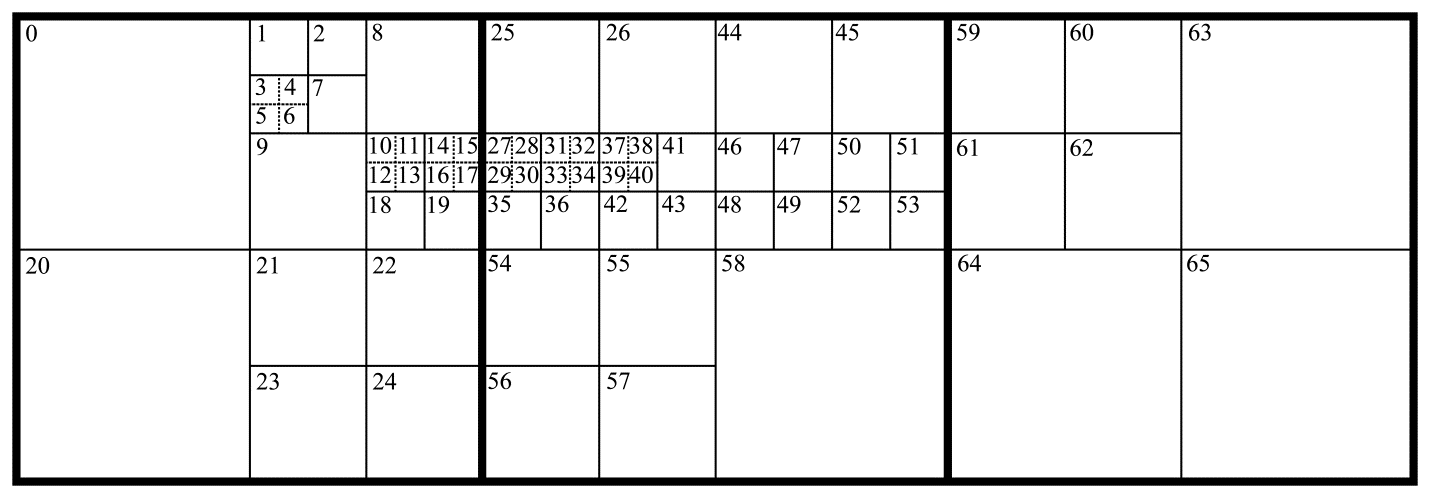


Рис. 3 Положение блоков при разбиении кадра на квадродерево

При пространственном предсказании значений пикселов внутри текущего кодируемого блока в HEVC используются значения пикселов блоков, соседних с кодируемым – референсных (рис. 4). При этом референсные пикселы могут быть не доступны для кодирования, например, для блока D на рис. 4 недоступен ни один пиксел. Также в HEVC разрешено использовать для предсказания только те референсные пикселы, которые относятся к уже закодированным блокам, что позволяет однозначно декодировать кадр на стороне клиента. При этом, если в данный момент кодируется блок 7 (рис. 3), для кодирования не будут доступны референсные пикселы блоков 8 и 9. Данная проблема решается следующими правилами: все недоступные значения приравниваются значению первого доступного отсчета. Так, например, если недоступны блоки D, A, B (рис. 4), то значения всех референсных пикселов из этих блоков устанавливаются равными значению самого верхнего референсного отсчета из блока E. Если недоступен блок B, то вся линейка значений отсчетов этого блока устанавливается равной значению самого правого референсного пиксела из блока A. Этап генерации для вертикальной линейки недоступных референсных отсчетов выполняется симметрично. Наконец, в том случае, когда ни один из блоков A, B, D, E, F недоступен, значения всех референсных отсчетов устанавливаются равными половине от максимально возможного значения. То есть, если кодируется видео, в котором отсчеты представляются 8-разрядными числами, то в этом случае всем референсным пикселам присваивается значение 127. [6]

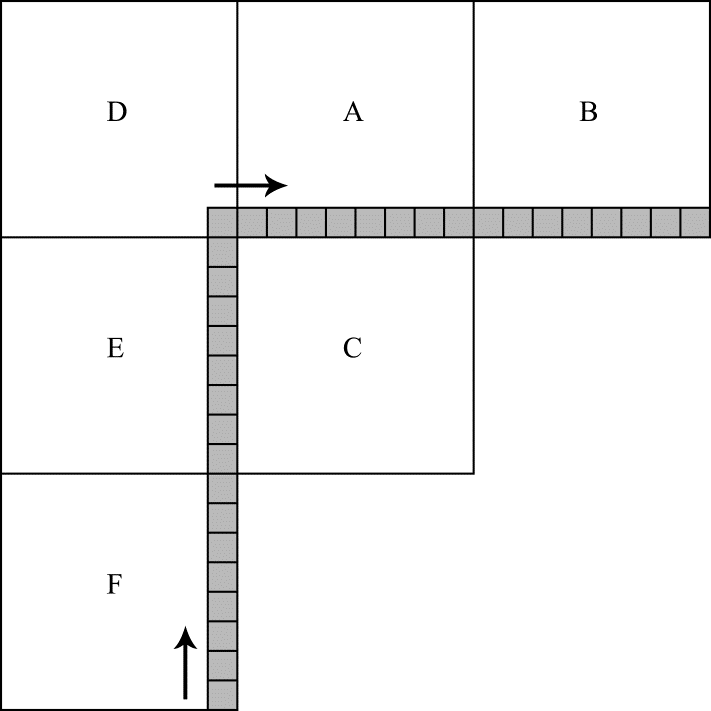


Рис. 4 Положение и порядок обхода референсных пикселов

Пространственное кодирование можно разделить на четыре этапа.

На первом этапе выполняется генерация недоступных для кодирования пикселов как показано выше. Этот этап выполняется каждый раз, когда недоступные пикселы есть.

На втором этапе предсказания последовательность референсных отсчетов, перебираемых сначала снизу-вверх (то есть начиная с нижнего отсчета в блоке F и заканчивая отсчетом в блоке D), а затем слева-направо (то есть начиная с левого отсчета в блоке A и заканчивая самым правым отсчетом в блоке B), подвергается фильтрации. Направление перебора отсчетов при фильтрации отмечено на рис. 4 стрелками. Тип фильтра определяется размерами кодируемого блока.

На третьем этапе производится расчет значений пикселов внутри кодируемого блока. HEVС определяет 35 режимов расчета пикселов:

режим 0 - DC режим при котором значения пикселов блока усредняется;

режим 1 - Planar режим при котором значения пикселов блока получается в результате билинейной интерполяции с учетом референсных пикселов. Значение каждого пиксела рассчитывается по формуле:

 (1),

где  – размер блока, x, y – координаты пикселов, , , - значения референсных пикселов [5].

режимы 2 – 32 – Angular режимы (рис. 5). В этих режимах значения референсных пикселов в заданном направлении расставляются внутри кодируемого блока. В том случае, когда позиция предсказываемого пиксела  попадает между референсными отсчетами, в качестве предсказания используется интерполированное значение. Для режимов с номерами от 2 до 17 расстановка референсных значений производится в направлениях слева-направо. Для режимов с номерами от 18 до 34 такая расстановка ведется сверху-вниз. Сами режимы отличаются друг от друга только выбором угла *φ*, на основе которого выбираются референсные пикселы.

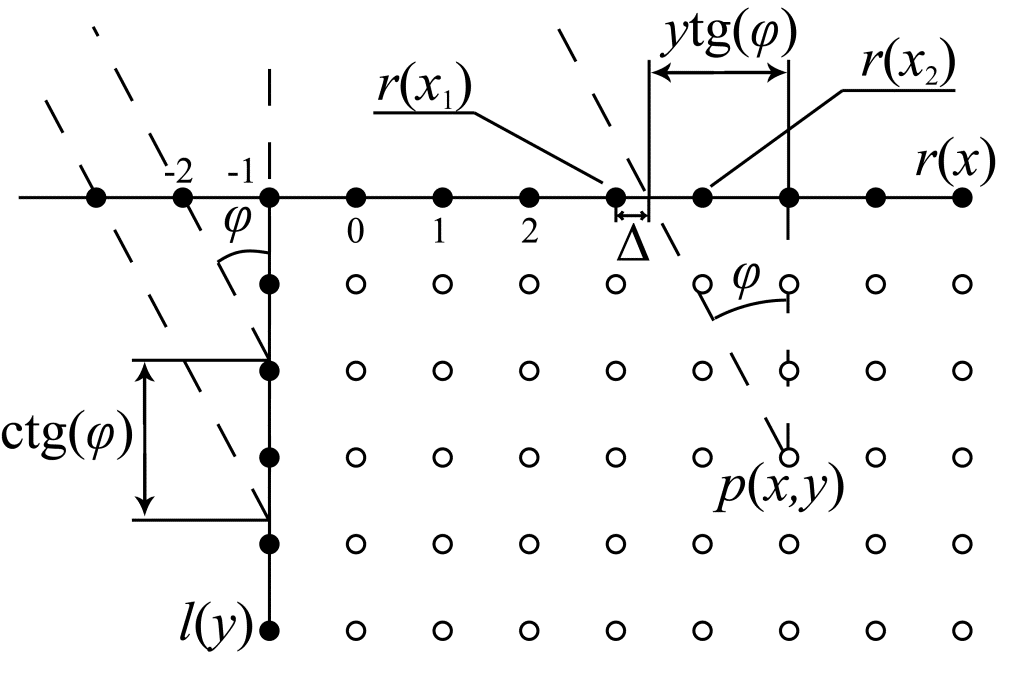


Рис. 5 Внетрикадровое кодирование в Angular режимах

1. Практический раздел

Для демонстрации работы алгоритмов кодирования и декодирования изображения была разработана программа, реализующая передачу информации по протоколу UDP. Диаграмма классов данной программы приведена на рис. 6.

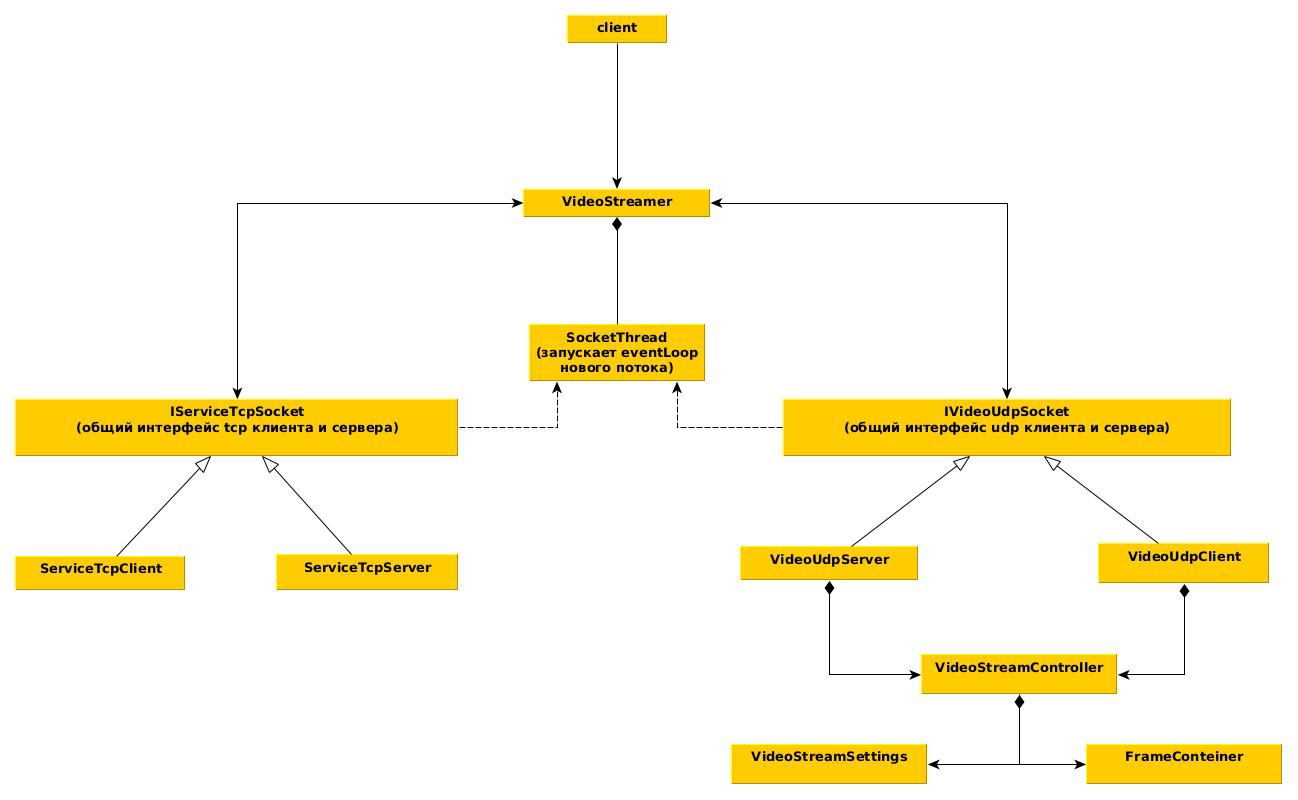


Рис. 6 Диаграмма классов приложения, реализующего передачу данных по протоколу UDP

**Описание классов.**

Класс VideoStreamer предоставляет пользователю интерфейс для работы с программой, а именно: запуск/остановку передачи/приема видеопотока, настройки хоста (ip-адрес и порт), настройку частоты кадров при захвате изображений с экрана, максимальный размер одного пакета данных при передаче по UDP, задержка передачи. Также данный класс является виджетом Qt framework для отрисовки полученных кадров видеопотока.

Класс IServiceTcpSocker является абстрактным TCP сокетом, который предоставляет общий интерфейс для TCP клиента и TCP сервера

Класс ServiceTcpClient реализует интерфейс ISecviceTcpSocket и предоставляет возможность отправлять сервисные сообщения о статусе передачи данных по протоколу TCP.

Класс ServiceTcpServer реализует интерфейс IServiceTcpSocket и предоставляет возможность отправлять сервисные сообщения о статусе передачи данных по протоколу TCP.

Класс IVideoUdpSocket является абстрактным UDP сокетом, который предоставляет общий интерфейс для UDP клиента и UDP сервера.

Класс VideoUdpClient реализует интерфейс IVideoUdpSocket, управляет передачей пакетов по протоколу UDP.

Класс VideoUdpServer реализует интерфейс IVideoUdpSocket, управляет приемом пакетов по протоколу UDP.

Класс VideoStreamController управляет сборкой пакетов из видеокадра для отправки по UDP протоколу и сборкой видеокадров из полученных по UDP протоколу пакетов используя алгоритмы кодирования и декодирования изображений.

Класс VideoStreamSettings хранит настройки соединения, а также параметры кодирования и декодирования изображений.

Класс FrameContainer содержит данные об изображении, а также реализует алгоритмы кодирования и декодирования. Может быть получен из собранных классом VideoStreamController пакетов в виде класса QByteArray, а также из захваченного с экрана изображения в виде класса QImage. Также реализует обратную конвертацию в вышеуказанные классы.

Класс SocketThread контролирует выделенные для видеозахвата, передачи и приема пакетов потоки.

**Реализация алгоритмов кодирования и декодирования изображений**

Так как протокол UDP является протоколом передачей данных с потерями, т.е. протоколом не гарантируется, что пакет n будет доставлен в ста процентах случаев, а алгоритм межкадрового уменьшения избыточности требует хранения нескольких взаимосвязанных кадров, которые должны быть идентичными у кодера и декодера, было принято решение отказаться от реализации данного алгоритма и производить кодирование только алгоритмом внутрикадрового уменьшения избыточности.

Формирование и отправка пакетов данных реализована следующим образом:

После захвата изображения в виде класса QImage на его основе класс FrameContainer заполняет матрицу блоков размерами H/8xW/8, где H – высота изображения, W – ширина изображения. Каждый блок при этом является трехмерной матрицей 8х8х3 и хранит данные о пикселах. При этом, так как QImage предоставляет данные пикселов в цветовом пространстве RGB, при заполнении матриц блоков пикселами происходит преобразование в цветовое пространство YUV по следующим формулам:

Y = 0.299 \* r + 0.587 \* g + 0.114 \* b - 128;

Cb = 128 - 0.168736 \* r - 0.331264 \* g + 0.5 \* b - 128;

Cr = 128 + 0.5 \* r - 0.418688 \* g - 0.081312 \* b - 128;

где Y – яркость, Cb, Cr – цветоразностные компоненты, R, G, B – красная, зеленая и синяя компоненты пиксела соответственно. Полученный экземпляр класса FrameContainer попадает в очередь на обработку, длинна которой задается при создании класса VideoStreamController.

При получении сигнала после начала формирования новых пакетов для отправки, класс VideoStreamController получает новый экземпляр FrameContainer из очереди кадров, запуская после этого алгоритм внутрикадрового предсказания полученного кадра.

По окончанию алгоритма экземпляром класса FrameContainer формируется пакет для отправки.

Декодирование сообщения осуществляется в обратном порядке, при этом при получении экземпляра QImage цветовое пространство YUV преобразуется в цветовое пространство RGB по следующим формулам:

r = Y + 128 + 1.402 \* (Cr);

g = Y+ 128 - 0.34414 \* (Cb) - 0.71414 \* (Cr);

b = Y + 128 + 1.772 \* (Cb);

**Псевдокод алгоритма внутрикадрового предсказания**

Алгоритм кодирования внутрикадрового предсказания реализует функция void encodIennerPrediction(); класса FrameContainer. Так как в реализации данного алгоритма не используется адаптивное разбиение на блоки (размер блока фиксирован и равен 8х8) недоступными оказываются только пикселы левее первого столбца и выше первой строки, что исключает необходимость проверки блоков ниже и правее кодируемого.

Псевдокод:

Создается копия класса FrameContainer f

Для каждой строки блоков 8х8 y

Для каждого блока 8х8 в строке x

Объявим массивы с референсными покселами xPixels[8], yPixels[9]

Если данная строка блоков первая и данный столбец блоков первый

Записываем в левый верхний референсный пиксел yPixels[0] среднее значение

Иначе

Записываем в левый верхний референсный пиксел yPixels[0] значение нижнего правого пиксела из блока из данного блока (f[y-1][x-1].pixels[7][7])

Для всех каждого незаписанного референсного пиксела i

Если блок х находится левее первого столбца

Записываем в референсный пиксел yPixels[i+1] среднее значение

Иначе

Записываем в референсный пиксел yPixels[i+i] значение пиксела f[y][x-1].pixels[i][7]

Если блок y находится выше первой строки

Записываем в референсный пиксел xPixels[i] среднее значение

Иначе

Записываем в референсный пиксел yPixels[i] значение пиксела f[y-1][x].pixels[7][i]

Для всех строк пикселов в блоке 8х8 by

Для всех пикселов в строке bx

По формуле 1 получаем значение интерполированных компонент пиксела

Находим разницу между текущим значением данного пиксела и полученным его интерполированным значением

Заменяем текущий пиксел полученной разностью

Блок схема алгоритма приведена на рис. 6.



Рис. 7 Блок-схема алгоритма внутрикадрового предсказания

1. Результаты

Как было указано выше результатом работы алгоритма является получение как можно большего количество значений компонент пикселов близких к нулю.

Ниже приведен блок из 23 строки 8 столбца матрицы блоков до и после кодирования с значениями компонент содержащихся в нем пикселов. Как можно увидеть после кодирования значения отдельных компонент пикселов либо равны нулю, либо колеблются около него, что позволяет после выполнения косинусного преобразования Фурье, квантования и энтропийного кодирования [5] получить достаточно высокую степень сжатия при выполнении кодирования для каждой плоскости компонент отдельно.

Блок [23][8] до кодирования

[58 -15 21][58 -15 21][59 -16 21][61 -15 21][63 -15 21][65 -15 21][67 -15 21][65 -15 22]

[59 -15 21][60 -15 21][60 -15 21][61 -15 21][63 -15 21][65 -15 21][66 -15 21][65 -15 21]

[61 -15 21][62 -15 21][62 -15 21][63 -15 21][65 -15 21][66 -15 21][67 -15 21][66 -15 21]

[64 -15 21][64 -15 21][65 -15 21][65 -15 21][65 -15 21][66 -15 21][67 -15 21][66 -15 22]

[63 -15 21][64 -15 21][64 -15 21][64 -15 21][64 -15 21][64 -15 21][64 -15 21][64 -15 22]

[62 -15 21][63 -15 21][63 -15 21][63 -15 21][62 -15 21][61 -15 21][61 -15 21][61 -15 22]

[62 -15 21][63 -15 21][63 -15 21][62 -15 21][62 -15 21][60 -15 21][59 -15 21][59 -15 22]

[63 -15 21][64 -15 21][64 -15 21][64 -15 21][62 -15 21][61 -15 21][60 -15 21][60 -15 22]

Блок [23][8] после кодирования

[-2 -1 0][ 0 -1 0][ 2 -1 0][ 2 -1 0][ 2 -1 0][ 3 -1 0][ 4 -1 -1][ 5 -1 -1]

[-3 -1 0][-1 -1 0][ 2 -1 0][ 1 -1 0][ 1 -1 0][ 4 -1 0][ 4 -1 0][ 4 -1 -1]

[-3 -1 0][-3 -1 0][-3 -1 0][-4 -1 0][-2 -1 0][ 0 -1 0][ 2 -1 0][ 1 -1 0]

[-1 -1 0][-3 -1 1][-3 -1 0][-3 -1 1][ 0 -2 1][ 1 -2 1][ 2 -1 1][ 1 -1 1]

[-1 -1 0][-2 -1 0][-2 -2 0][-1 -1 1][ 1 -1 1][ 2 -1 1][ 3 -1 1][ 1 -1 2]

[-1 -1 0][-1 -1 0][-2 -1 0][-1 -1 0][ 1 -1 1][ 2 -1 1][ 2 -1 1][ 1 -1 1]

[ 0 -1 0][ 1 -1 0][ 0 -1 0][ 1 -1 0][ 3 -1 0][ 3 -1 1][ 3 -1 1][ 2 -1 1]

[ 2 -1 0][ 2 -1 0][ 2 -1 0][ 2 -1 0][ 2 -1 0][ 3 -1 0][ 4 -1 1][ 2 -1 2]

1. Выводы

В ходе производственной практики был реализован алгоритм внутрикадрового предсказания, позволяющий уменьшить требования к пропускной способности канала при потоковой передаче видео.

Из плюсов данного алгоритма можно отметить хорошую степень сжатия данных после преобразования Фурье, квантования и энтропийного кодирования в следствие получение разностного сигнала колеблющегося около ноля, а также отсутствие необходимости хранения дополнительных связанных с кодируемым кадров, что позволяет передавать данные по протоколу UDP и, как следствие, избежать переполнения буфера при разрыве соединения, как если бы использовался протокол TCP.

Из минусов данного алгоритма можно выделить большую избыточность в сравнении с алгоритмом межкадрового предсказания при передаче кадров в статических сценах, а также низкую степень сжатия при кодировании изображений, содержащих большое количество мелких объектов различных цветов в кадре.

Для дальнейшего улучшения алгоритма необходимо организовать адаптивное деление на блоки, что позволит добиться большей степени сжатия для кадров с большим содержанием мелких объектов различных цветов. Так же необходима фильтрация кадра после декодирования, что позволит уменьшить отфильтровать потери при квантовании, а также позволит применить систему цветовой субдискретизации 4:2:2 или 4:0:0 [5].

# Приложение 1

Исходный код класса FrameContainer

Файл FrameContainer.h

#ifndef FRAMECONTAINER\_H

#define FRAMECONTAINER\_H

#include <QImage>

#include <QHash>

class FrameContainer

{

//! чтобы не создавать статических, глобальных или константных переменных

//! в будущем поможет создать адаптивное разбиение при пространственном преобразовании

enum sizes {

BLOCK = 8,

PIXEL = 3,

};

//! Содержит данные отдельного пиксела

//! Y - unsingedPix[0]

//! Cb - unsingedPix[1]

//! Cr - pix.unsingedPix[2]

// struct Pixel { quint8 pix[PIXEL]; };

//! Чтобы получить симметрию относительно 0

struct Pixel { qint8 pix[PIXEL]; };

//! Содержит данные отдельного блока пикселов

struct Block { Pixel pixels[BLOCK][BLOCK]; };

public:

static void fillTables();

FrameContainer(int height, int width);

//! Так как есть распределенная память

//! @{

~FrameContainer();

FrameContainer(const FrameContainer &other);

FrameContainer operator=(const FrameContainer &other);

//! @}

//! false, если контейнер нельзя использовать.

//! Любое использование контейнера при этом может вызовать непредвиденное поедение

bool isValid() const;

//! Создает контейнер из извображения

static FrameContainer fromImage(QImage img);

//! Когда будет доступно арифметическое кодирование или кодирование Хаффмана, это будет удалено

//! @{

static FrameContainer fromByteArray(QByteArray ba, int imgWidth, int imgHeight);

QByteArray toByteArray();

//! @}

//! Кодирование внетрекадрового предсказания

void encodIennerPrediction();

//! Декодирование внутрекадрового предсказания

void decodeInnerPrediction();

//! Создает изображение на основе данных контейнера.

//! Данные должны быть полностью декадированы

QImage toQImage();

void out(const QString &filename, int numb);

private:

Block \*\*m\_blocks;

int m\_xBlocksCount;

int m\_yBlocksCount;

int m\_width;

int m\_height;

static QHash<quint8, QHash<quint8, quint8>> m\_lpsTab;

static QHash<quint8, QPair<quint8, quint8>> m\_transIdxMap;

};

#endif // FRAMECONTAINER\_H

Файл FrameContainer.cpp

#include "framecontainer.h"

#include <QDebug>

#include <QFile>

QHash<quint8, QHash<quint8, quint8>> FrameContainer::m\_lpsTab;

QHash<quint8, QPair<quint8, quint8>> FrameContainer::m\_transIdxMap;

FrameContainer::FrameContainer(int height, int width)

: m\_blocks(Q\_NULLPTR)

, m\_xBlocksCount(width / BLOCK)

, m\_yBlocksCount(height / BLOCK)

, m\_width(width)

, m\_height(height)

{

if(height != 0 && width != 0) {

m\_blocks = new Block\*[m\_yBlocksCount];

for(int i = 0; i < m\_yBlocksCount; ++i)

m\_blocks[i] = new Block[m\_xBlocksCount];

}

}

FrameContainer::~FrameContainer()

{

if(isValid()) {

for(int i = 0; i < m\_yBlocksCount; ++i)

delete[] m\_blocks[i];

delete [] m\_blocks;

}

}

FrameContainer::FrameContainer(const FrameContainer &other)

: m\_blocks(Q\_NULLPTR)

, m\_xBlocksCount(other.m\_xBlocksCount)

, m\_yBlocksCount(other.m\_yBlocksCount)

, m\_width(other.m\_width)

, m\_height(other.m\_height)

{

if(other.m\_height != 0 && other.m\_width != 0) {

m\_blocks = new Block\*[m\_yBlocksCount];

for(int i = 0; i < m\_yBlocksCount; ++i) {

m\_blocks[i] = new Block[m\_xBlocksCount];

for(int j = 0; j < m\_xBlocksCount; ++j)

m\_blocks[i][j] = other.m\_blocks[i][j];

}

}

}

FrameContainer FrameContainer::operator=(const FrameContainer &other)

{

if(&other == this)

return \*this;

m\_height = other.m\_height;

m\_width = other.m\_width;

m\_xBlocksCount = other.m\_xBlocksCount;

m\_yBlocksCount = other.m\_yBlocksCount;

for(int i = 0; i < m\_yBlocksCount; ++i)

delete[] m\_blocks[i];

delete[] m\_blocks;

m\_blocks = Q\_NULLPTR;

if(other.m\_height != 0 && other.m\_width != 0) {

m\_blocks = new Block\*[m\_yBlocksCount];

for(int i = 0; i < m\_yBlocksCount; ++i) {

m\_blocks[i] = new Block[m\_xBlocksCount];

for(int j = 0; j < m\_xBlocksCount; ++j)

m\_blocks[i][j] = other.m\_blocks[i][j];

}

}

return \*this;

}

bool FrameContainer::isValid() const

{

return m\_width != 0 && m\_height != 0 && m\_blocks != Q\_NULLPTR;

}

FrameContainer FrameContainer::fromImage(QImage img)

{

if(img.isNull())

return FrameContainer(0, 0);

FrameContainer container(img.height(), img.width());

qint16 r = 0;

qint16 g = 0;

qint16 b = 0;

qint8 Cb = 0;

qint8 Cr = 0;

for(int y = 0; y < container.m\_yBlocksCount; ++y) {

for(int x = 0; x < container.m\_xBlocksCount; ++x) {

Block &block = container.m\_blocks[y][x];

for(int by = 0; by < BLOCK; ++by) {

const QRgb \*imgLine = reinterpret\_cast<QRgb \*>(img.scanLine(y \* BLOCK + by));

for(int bx = 0; bx < BLOCK; ++bx) {

const QRgb &imgPix = imgLine[x \* BLOCK + bx];

Pixel &pix = block.pixels[by][bx];

r = qRed(imgPix);

g = qGreen(imgPix);

b = qBlue(imgPix);

//Y

pix.pix[0] = 0.299 \* r + 0.587 \* g + 0.114 \* b - 128;

// if(bx % 2 == 0) {

Cb = 128 - 0.168736 \* r - 0.331264 \* g + 0.5 \* b - 128;

Cr = 128 + 0.5 \* r - 0.418688 \* g - 0.081312 \* b - 128;

// }

//Cb

pix.pix[1] = Cb;

//Cr

pix.pix[2] = Cr;

}

}

}

}

return container;

}

FrameContainer FrameContainer::fromByteArray(QByteArray arr, int imgWidth, int imgHeight)

{

QDataStream stream(&arr, QIODevice::ReadOnly);

FrameContainer container(imgHeight, imgWidth);

quint8 Cb = 0;

quint8 Cr = 0;

for(int y = 0; y < container.m\_yBlocksCount; ++y) {

for(int x = 0; x < container.m\_xBlocksCount; ++x) {

Block &block = container.m\_blocks[y][x];

for(int by = 0; by < BLOCK; ++by) {

for(int bx = 0; bx < BLOCK; ++bx) {

Pixel &pix = block.pixels[by][bx];

stream >> pix.pix[0];

stream >> Cb >> Cr;

pix.pix[1] = Cb;

pix.pix[2] = Cr;

}

}

}

}

return container;

}

void FrameContainer::encodIennerPrediction()

{

Pixel avaragePix;

avaragePix.pix[0] = 127;

avaragePix.pix[1] = 127;

avaragePix.pix[2] = 127;

FrameContainer uncodedFrame = \*this;

for(int y = 0; y < m\_yBlocksCount; ++y) {

for(int x = 0; x < m\_xBlocksCount; ++x) {

Pixel xPixels[8];

Pixel yPixels[9];

bool canCodeYBlock = false;

bool canConeXBlock = false;

// Получить верхний левый референсный пиксел

Pixel &yPix = yPixels[0];

if(x == 0 || y == 0)

yPix = avaragePix;

else

yPix = uncodedFrame.m\_blocks[y - 1][x - 1].pixels[BLOCK - 1][BLOCK - 1];

// Получаем референсные пикселы слева и сверху от текущего блока

for(int r\_1 = 0; r\_1 < BLOCK; ++r\_1) {

canCodeYBlock = x != 0;

canConeXBlock = y != 0;

Pixel &yPix = yPixels[r\_1 + 1];

Pixel &xPix = xPixels[r\_1];

if(canCodeYBlock)

yPix = uncodedFrame.m\_blocks[y][x - 1].pixels[r\_1][BLOCK - 1];

else {

yPix = xPixels[0];

}

if(canConeXBlock)

xPix = uncodedFrame.m\_blocks[y - 1][x].pixels[BLOCK - 1][r\_1];

else {

xPix = yPixels[0];

}

}

Block &block = m\_blocks[y][x];

for(int by = 0; by < BLOCK; ++by) {

for(int bx = 0; bx < BLOCK; ++bx) {

Pixel delta;

Pixel &pix = block.pixels[by][bx];

delta.pix[0] = ((BLOCK - 1 - bx) \* (yPixels[by + 1].pix[0])

+ (bx + 1) \* (xPixels[BLOCK - 1].pix[0])

+ (BLOCK - 1 - by) \* (xPixels[bx].pix[0])

+ (by + 1) \* (yPixels[BLOCK].pix[0]) + (BLOCK)) / (2 \* BLOCK);

delta.pix[1] = ((BLOCK - 1 - bx) \* (yPixels[by + 1].pix[1])

+ (bx + 1) \* (xPixels[BLOCK - 1].pix[1])

+ (BLOCK - 1 - by) \* (xPixels[bx].pix[1])

+ (by + 1) \* (yPixels[BLOCK].pix[1]) + (BLOCK)) / (2 \* BLOCK);

delta.pix[2] = ((BLOCK - 1 - bx) \* (yPixels[by + 1].pix[2])

+ (bx + 1) \* (xPixels[BLOCK - 1].pix[2])

+ (BLOCK - 1 - by) \* (xPixels[bx].pix[2])

+ (by + 1) \* (yPixels[BLOCK].pix[2]) + (BLOCK)) / (2 \* BLOCK);

pix.pix[0] -= delta.pix[0];

pix.pix[1] -= delta.pix[1];

pix.pix[2] -= delta.pix[2];

}

}

}

}

}

void FrameContainer::decodeInnerPrediction()

{

Pixel avaragePix;

avaragePix.pix[0] = 127;

avaragePix.pix[1] = 127;

avaragePix.pix[2] = 127;

for(int y = 0; y < m\_yBlocksCount; ++y) {

for(int x = 0; x < m\_xBlocksCount; ++x) {

Pixel xPixels[8];

Pixel yPixels[9];

bool canCodeYBlock = false;

bool canConeXBlock = false;

// Получить верхний левый референсный пиксел

Pixel &yPix = yPixels[0];

if(x == 0 || y == 0)

yPix = avaragePix;

else

yPix = m\_blocks[y - 1][x - 1].pixels[BLOCK - 1][BLOCK - 1];

// Получаем референсные пикселы слева и сверху от текущего блока

for(int r\_1 = 0; r\_1 < BLOCK; ++r\_1) {

canCodeYBlock = x != 0;

canConeXBlock = y != 0;

Pixel &yPix = yPixels[r\_1 + 1];

Pixel &xPix = xPixels[r\_1];

if(canCodeYBlock)

yPix = m\_blocks[y][x - 1].pixels[r\_1][BLOCK - 1];

else {

yPix = xPixels[0];

}

if(canConeXBlock)

xPix = m\_blocks[y - 1][x].pixels[BLOCK - 1][r\_1];

else {

xPix = yPixels[0];

}

}

Block &block = m\_blocks[y][x];

for(int by = 0; by < BLOCK; ++by) {

for(int bx = 0; bx < BLOCK; ++bx) {

Pixel delta;

Pixel &pix = block.pixels[by][bx];

delta.pix[0] = ((BLOCK - 1 - bx) \* (yPixels[by + 1].pix[0])

+ (bx + 1) \* (xPixels[BLOCK - 1].pix[0])

+ (BLOCK - 1 - by) \* (xPixels[bx].pix[0])

+ (by + 1) \* (yPixels[BLOCK].pix[0]) + (BLOCK)) / (2 \* BLOCK);

delta.pix[1] = ((BLOCK - 1 - bx) \* (yPixels[by + 1].pix[1])

+ (bx + 1) \* (xPixels[BLOCK - 1].pix[1])

+ (BLOCK - 1 - by) \* (xPixels[bx].pix[1])

+ (by + 1) \* (yPixels[BLOCK].pix[1]) + (BLOCK)) / (2 \* BLOCK);

delta.pix[2] = ((BLOCK - 1 - bx) \* (yPixels[by + 1].pix[2])

+ (bx + 1) \* (xPixels[BLOCK - 1].pix[2])

+ (BLOCK - 1 - by) \* (xPixels[bx].pix[2])

+ (by + 1) \* (yPixels[BLOCK].pix[2]) + (BLOCK)) / (2 \* BLOCK);

pix.pix[0] += delta.pix[0];

pix.pix[1] += delta.pix[1];

pix.pix[2] += delta.pix[2];

}

}

}

}

}

QByteArray FrameContainer::toByteArray()

{

if(!isValid())

return QByteArray();

QByteArray ba;

QDataStream stream(&ba, QIODevice::WriteOnly);

for(int y = 0; y < m\_yBlocksCount; ++y) {

for(int x = 0; x < m\_xBlocksCount; ++x) {

const Block &block = m\_blocks[y][x];

for(int by = 0; by < BLOCK; ++by) {

for(int bx = 0; bx < BLOCK; ++bx) {

const Pixel &pix = block.pixels[by][bx];

stream << pix.pix[0];

stream << pix.pix[1];

stream << pix.pix[2];

}

}

}

}

return ba;

}

QImage FrameContainer::toQImage()

{

if(!isValid())

return QImage();

QImage img(m\_width, m\_height, QImage::Format\_RGB32);

quint16 r = 0;

quint16 g = 0;

quint16 b = 0;

for(int y = 0; y < m\_yBlocksCount; ++y) {

for(int x = 0; x < m\_xBlocksCount; ++x) {

const Block &block = m\_blocks[y][x];

for(int by = 0; by < BLOCK; ++by) {

QRgb \*imgLine = reinterpret\_cast<QRgb \*>(img.scanLine(y \* BLOCK + by));

for(int bx = 0; bx < BLOCK; ++bx) {

QRgb &imgPix = imgLine[x \* BLOCK + bx];

const Pixel &pix = block.pixels[by][bx];

r = pix.pix[0] + 128 + 1.402 \* (pix.pix[2]);

g = pix.pix[0] + 128 - 0.34414 \* (pix.pix[1]) - 0.71414 \* (pix.pix[2]);

b = pix.pix[0] + 128 + 1.772 \* (pix.pix[1]);

if(r > 255) r = 255;

else if(r < 0) r = 0;

if(g > 255) g = 255;

else if(g < 0) g = 0;

if(b > 255) b = 255;

else if(b < 0) b = 0;

imgPix = qRgb(r, g, b);

}

}

}

}

return img;

}

void FrameContainer::out(const QString &filename, int numb)

{

if(!isValid())

return;

QFile outFile(filename);

if(!outFile.open(QIODevice::WriteOnly)) {

qDebug() << "cannot open file" << filename;

return;

}

QTextStream stream(&outFile);

stream << numb << '\n';

for(int y = 0; y < m\_yBlocksCount; ++y) {

for(int by = 0; by < BLOCK; ++by) {

for(int x = 0; x < m\_xBlocksCount; ++x) {

for(int bx = 0; bx < BLOCK; ++bx) {

Pixel &pix = m\_blocks[y][x].pixels[by][bx];

stream << "[" << pix.pix[0] << ' ' << pix.pix[1] << ' ' << pix.pix[2] << "]";

}

}

stream << '\n';

}

}

}

Список литературы

1. Дворкович, В. П. Цифровые видеоинформационные системы (тория и практика) / В. П. Дворкович, А. В. Дворкович, М.: ТЕХНОСФЕРА, 2012 – 1008 с.
2. Ян Ричардсон. Видеокодирование. Н.264 и MPEG-4 — стандарты нового поколения Москва: Техносфера, 2005. - 368 с.
3. Патент № 2273113 Российская Федерация, МПК H04N 7/36, H04N 1/417, H04N 5/937. Способ кодирования движущегося изображения / Джеон Байеонг-Мун. У. – 2003. – 13 с.
4. Пономарев О.Г. Критерий для быстрого выбора размера блока  
   пространственного предсказания в системе видеокодирования  
   HEVC / О.Г. Пономарев, М.П. Шарабайко, Д.Ю. Тё // Доклады  
   ТУСУРа. 2015.– № 3. – С. 106–113.
5. Recommendation ITU-T H.265 | International Standart ISO/IES 23008-2 / International Telecommunication Union – G., 2019
6. Wien, Mathias High Efficiency Video Coding – Coding Tools and Specification / Wien, Mathias, B.: Springer-Verlag, 2015 - 314